

NAVAL POSTGRADUATE SCHOOL

Monterey, California



DISSERTATION

**HOLISTIC FRAMEWORK FOR ESTABLISHING
INTEROPERABILITY OF HETEROGENEOUS
SOFTWARE DEVELOPMENT TOOLS**

by

Joseph F. Puett III

June 2003

Dissertation Supervisor:

Luqi

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE June 2003	3. REPORT TYPE AND DATES COVERED Dissertation	
4. TITLE AND SUBTITLE: Holistic Framework For Establishing Interoperability of Heterogeneous Software Development Tools			5. FUNDING NUMBERS	
6. AUTHOR(S) Joseph F. Puett III				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Space and Naval Warfare Systems Center- San Diego San Diego, CA 92152-5031			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) <p>This dissertation presents a Holistic Framework for Software Engineering (HFSE) that establishes collaborative mechanisms by which existing heterogeneous software development tools and models will interoperate. Past research has been conducted with the aim of developing or improving individual aspects of software development; however, this research focuses on establishing a holistic approach over the entire development effort where unrealized synergies and dependencies between all of the tools' artifacts can be visualized and leveraged to produce both improvements in process and product.</p> <p>The HFSE is both a conceptual framework and a software engineering process model (with tool support) where the dependencies between software development artifacts are identified, quantified, tracked, and deployed throughout all artifacts via middleware. Central to the approach is the integration of Quality Function Deployment (QFD) into the Relational Hypergraph (RH) Model of Software Evolution. This integration allows for the dependencies between artifacts to be automatically tracked throughout the hypergraph representation of the development effort, thus assisting the software engineer to isolate subgraphs as needed.</p>				
14. SUBJECT TERMS Software Evolution, Interoperability, Integrated Software Development Environments, Heterogeneous Software Systems, Quality Function Deployment			15. NUMBER OF PAGES 370	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

**HOLISTIC FRAMEWORK FOR ESTABLISHING INTEROPERABILITY OF
HETEROGENEOUS SOFTWARE DEVELOPMENT TOOLS**

Joseph F. Puett III
Lieutenant Colonel, United States Army
B.S., United States Military Academy, 1982
M.S., California Institute of Technology, 1992
M.B.A., Long Island University, 1994

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN SOFTWARE ENGINEERING

from the

**NAVAL POSTGRADUATE SCHOOL
June 2003**

Author:

Joseph F. Puett III

Approved by:

Luqi
Professor of Computer Science
Dissertation Supervisor

Craig W. Rasmussen
Associate Professor of Mathematics

James B. Michael
Associate Professor of
Computer Science

Man-Tak Shing
Associate Professor of
Computer Science

Nelson Ludlow
Mobilisa, Incorporated

Approved by:

Peter Denning, Chair, Department of Computer Science

Approved by:

Carson K. Eoyang, Associate Provost for Academic Affairs

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This dissertation presents a Holistic Framework for Software Engineering (HFSE) that establishes collaborative mechanisms by which existing heterogeneous software development tools and models will interoperate. Past research has been conducted with the aim of developing or improving individual aspects of software development; however, this research focuses on establishing a holistic approach over the entire development effort where unrealized synergies and dependencies between all of the tools' artifacts can be visualized and leveraged to produce both improvements in process and product.

The HFSE is both a conceptual framework and a software engineering process model (with tool support) where the dependencies between software development artifacts are identified, quantified, tracked, and deployed throughout all artifacts via middleware. Central to the approach is the integration of Quality Function Deployment (QFD) into the Relational Hypergraph (RH) Model of Software Evolution. This integration allows for the dependencies between artifacts to be automatically tracked throughout the hypergraph representation of the development effort, thus assisting the software engineer to isolate subgraphs as needed.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	MOTIVATION	1
1.	Software Understandability	1
a.	<i>Produce Tools to Promote Understandability of Customer Desires and Legacy Software Development Efforts.....</i>	<i>2</i>
b.	<i>Produce a Framework that Promotes Understandability of Customer Desires and Legacy Software Development Efforts.....</i>	<i>3</i>
2.	Holistic Development.....	3
3.	Requirements Engineering.....	5
4.	Coherent Development	7
5.	Software Safety.....	8
6.	Solving these Problems with the Holistic Framework For Software Engineering (HFSE)	9
B.	RESEARCH HYPOTHESIS AND METHODOLOGY	10
1.	Research Hypothesis.....	10
2.	Research Methodology	10
a.	<i>Development of a Software Tool Artifact Ontology.....</i>	<i>11</i>
b.	<i>Integration of QFD and the Evolution Model.....</i>	<i>11</i>
c.	<i>Creation of a Federation Interoperability Object Model.....</i>	<i>12</i>
d.	<i>Prototype the HFSE by Extending CASES.....</i>	<i>12</i>
e.	<i>Application of the HFSE to a Software Development Scenario.....</i>	<i>13</i>
C.	OVERVIEW OF THE HOLISTIC FRAMEWORK FOR SOFTWARE EVOLUTION	13
1.	Software Evolution.....	14
2.	Object Model	16
3.	The Ideal HFSE.....	17
a.	<i>Generic</i>	<i>17</i>
b.	<i>Real Tools.....</i>	<i>17</i>
c.	<i>Process Independent</i>	<i>18</i>
d.	<i>Domain Independent.....</i>	<i>18</i>
e.	<i>Extensible</i>	<i>18</i>
f.	<i>Improve Time to Market</i>	<i>19</i>
g.	<i>Decrease Cost of Development</i>	<i>19</i>
h.	<i>Improved Quality.....</i>	<i>19</i>
i.	<i>Easy to Use and Enhances Productivity.....</i>	<i>19</i>
4.	Scope of the Dissertation Research	19
D.	CONTRIBUTIONS PROVIDED BY THIS DISSERTATION	19
1.	Accomplishment of the Research Goal	19
a.	<i>Construction of the HFSE is Feasible</i>	<i>19</i>
b.	<i>HFSE Artifacts Can be Described Mathematically.....</i>	<i>20</i>

c.	<i>The HFSE Increases Software Tool Interoperability.....</i>	<i>20</i>
2.	Other Original and Unique Contributions	20
a.	<i>Develop a Software Development Tool Ontology Construction Methodology</i>	<i>20</i>
b.	<i>Construct a Pilot Software Development Tool Ontology.....</i>	<i>20</i>
c.	<i>Adapt QFD Methodology to Deploy Software Dependencies other than Quality</i>	<i>20</i>
d.	<i>Apply OOMI to the Software Development Tool Domain.....</i>	<i>21</i>
e.	<i>Use the HFSE to Provide Perspective Views of the Development Effort.....</i>	<i>21</i>
E.	ORGANIZATION OF THE DISSERTATION	21
II.	PREVIOUS WORK.....	25
A.	CHAPTER ORGANIZATION.....	25
B.	FOUNDATION WORK.....	25
1.	Approaches to Software Evolution.....	25
a.	<i>Inevitability of Evolution</i>	<i>30</i>
b.	<i>Holistic Approach</i>	<i>31</i>
c.	<i>Assumptions and Uncertainty.....</i>	<i>31</i>
d.	<i>Validation Against Assumptions</i>	<i>31</i>
e.	<i>Views of Evolution</i>	<i>32</i>
2.	The Relational Hypergraph and CASES.....	32
a.	<i>Brief History of the Hypergraph</i>	<i>32</i>
b.	<i>The Relational Hypergraph (RH) Software Evolution Model.....</i>	<i>33</i>
3.	Quality Function Deployment (QFD).....	36
a.	<i>QFD History and Cited Benefits</i>	<i>36</i>
b.	<i>Software Quality Function Deployment (SQFD)</i>	<i>38</i>
c.	<i>The Voice of the Customer</i>	<i>41</i>
d.	<i>Steps in the QFD Process</i>	<i>45</i>
e.	<i>Adapting QFD to Software Development.....</i>	<i>49</i>
f.	<i>Establishing Correlations</i>	<i>55</i>
g.	<i>QFD in Large Complex Software Systems.....</i>	<i>57</i>
h.	<i>The Role of QFD in this Research</i>	<i>59</i>
4.	Methods for Establishing Interoperability of Software Development Models and Tools.....	60
5.	Application of Ontologies for Interoperability.....	63
a.	<i>Ontology Overview and Example</i>	<i>63</i>
b.	<i>Constructing an Ontology.....</i>	<i>69</i>
c.	<i>Ontology Definition and Capture.....</i>	<i>71</i>
d.	<i>UML as an Ontology Description Language.....</i>	<i>73</i>
C.	RELATED WORK.....	74
1.	Software Development Tool Suites: The Rational Approach.....	74
a.	<i>Summary.....</i>	<i>74</i>
b.	<i>Relationship to the Dissertation Topic</i>	<i>77</i>
c.	<i>Weaknesses.....</i>	<i>78</i>
2.	Rational Unified Process (RUP)	78

a.	<i>Summary</i>	79
b.	<i>Applicability or Relationship of Work to the Dissertation Topic</i>	81
c.	<i>Weaknesses</i>	81
3.	Integrated Software Development Environments	82
a.	<i>ISDEs and ISPEs</i>	82
b.	<i>Portable Common Tool Environment (PCTE)</i>	83
c.	<i>Arcadia</i>	84
d.	<i>Weaknesses</i>	84
4.	Software Engineering Ontologies	85
a.	<i>Software Engineering Body of Knowledge</i>	86
b.	<i>DARPA Agent Markup Language</i>	89
5.	The Uses of QFD for Software	90
D.	CHAPTER SUMMARY	92
III.	TOWARDS A SOFTWARE DEVELOPMENT TOOL ONTOLOGY	93
A.	CHAPTER OVERVIEW	93
B.	METHODOLOGY FOR BUILDING THE ONTOLOGY	93
1.	Step 1 -- Purpose and Scope of the Ontology	95
2.	Step 2 -- Feature Modeling	95
3.	Step 3 -- Establishing Commonalities	98
4.	Step 4 -- Tool Ontologies	99
5.	Step 5 - UML Representation of the Domain	100
6.	Step 6 -- Documentation	101
C.	DOMAIN ANALYSIS AND FEATURE MODELS	101
1.	Rational Requisite®Pro	102
2.	SEATools	107
D.	FEDERATION ONTOLOGY	112
E.	TOOL ONTOLOGIES	115
F.	ONTOLOGY INTER-RELATIONSHIPS	118
G.	CHAPTER SUMMARY	119
IV.	INTEGRATING QUALITY FUNCTION DEPLOYMENT INTO THE RELATIONAL HYPERGRAPH MODEL OF SOFTWARE EVOLUTION ..	121
A.	RELATIONAL HYPERGRAPH SOFTWARE EVOLUTION MODEL	121
1.	Overview of the Relational Hypergraph Software Evolution Model	121
2.	Important Definitions in the RH Model	121
3.	Embedding QFD within the Relational Hypergraph Software Evolution Model	124
a.	<i>Project Schema</i>	124
b.	<i>QFD Dependency</i>	124
B.	THE MATHEMATICS OF DEPENDENCY DEPLOYMENT	125
1.	Deployment Equations	125
2.	Downstream Dependency Deployment Example	128
3.	Upstream Deployment of Dependency	130
4.	Other Means of Deploying Dependencies: Constant Range	131

5.	Other Mathematical Checks	135
a.	<i>Superfluous Artifact Analysis</i>	135
b.	<i>Coverage Analysis</i>	136
C.	MAKING USE OF DEPLOYED DEPENDENCIES	138
1.	Dependency Threshold	138
2.	Component Trace.....	140
D.	ESTABLISHING DEPENDENCY VALUES	141
1.	Scales of Measurement	142
a.	<i>Nominal</i>	142
b.	<i>Ordinal</i>	142
c.	<i>Interval</i>	143
d.	<i>Ratio</i>	143
2.	QFD Dependency Valuation	144
a.	<i>Absolute Importance (Interval Valuation)</i>	144
b.	<i>Relative Importance (Ratio Valuation)</i>	145
c.	<i>Ordinal Importance (Ordinal Valuation)</i>	146
d.	<i>Comparison of Valuation Schemes</i>	146
3.	The Analytic Hierarchy Process (AHP).....	147
a.	<i>The Normalized Principal Eigenvector of Priority Values</i> ..	147
b.	<i>Consistency Checking of the Comparison Matrix</i>	149
c.	<i>Hierarchical Clustering to Account for Non-Independence</i>	149
4.	Subjectivity and Sensitivity Analysis	151
E.	CHAPTER SUMMARY.....	152
V.	APPLICATION OF THE OBJECT-ORIENTED METHODOLOGY FOR INTEROPERABILITY TO THE DOMAIN OF SOFTWARE DEVELOPMENT TOOLS	153
A.	CHAPTER OVERVIEW	153
B.	BUILDING A FEDERATION INTEROPERABILITY OBJECT MODEL	153
1.	Motivation for the FIOM	153
2.	FIOM Construction Methodology.....	155
C.	EXAMPLE OF CONSTRUCTING THE FIOM.....	157
D.	EXTENDING THE FIOM TO ACCOUNT FOR ADDITIONAL TOOLS.....	162
1.	Addition of Rational Rose® Example	162
2.	Addition of Pseudo-Code Example.....	163
E.	CREATING THE TOOL ADD-ONS.....	166
F.	LIMITATIONS OF THE OOMI APPROACH TO PROVIDING INTEROPERABILITY WITHIN THE HFSE	166
1.	The Intra-lingual Concept.....	167
2.	Scalability.....	167
3.	Ontologies and FIOMs are Difficult to Build	167
G.	SUMMARY	168
VI.	THE HOLISTIC FRAMEWORK FOR SOFTWARE ENGINEERING (HFSE)	169

A.	THE HFSE.....	169
B.	APPLYING THE HFSE TO ESTABLISH INTEROPERABILITY OF SOFTWARE ENGINEERING PROCESS MODELS	169
1.	Identifying The Project Schema	169
2.	Establishing the Tool Ontology	169
3.	Constructing the FIOM.....	170
4.	Establishing Communication Mechanisms.....	170
5.	Identifying Dependencies and Artifact Correlations.....	170
B.	EXTENSIONS.....	170
1.	Extending the FIOM with Additional Software Development Tools	170
2.	Extending the HFSE	171
3.	Adding Dependencies.....	171
C.	FOCUSED DEVELOPMENT USING "SLICES"	171
1.	Dependency Threshold	171
2.	Component Tracing.....	171
3.	Potential Application of Risk-Induced Slices	172
a.	<i>Greatest Risk Slices</i>	172
b.	<i>Change Knock-on Effects</i>	172
c.	<i>Safety Certification</i>	173
D.	CHAPTER SUMMARY.....	173
VII.	EXTENSIONS TO THE COMPUTER AIDED SOFTWARE EVOLUTION SYSTEM (CASES).....	175
A.	CHAPTER OVERVIEW	175
B.	GRAPHICALLY DEFINING A SOFTWARE DEVELOPMENT PROJECT SCHEMA	176
1.	The Project Schema in CASES version 1.1	176
2.	The Project Schema in CASES version 2.0	177
C.	EMBEDDING QFD INTO CASES	180
1.	QFD Dependencies.....	181
2.	Component Data Import	182
3.	QFD Matrices	184
D.	ENGINEERING VIEWS OF QFD DEPENDENCIES (SLICES OF THE RH MODEL).....	185
1.	Dependency Threshold View	186
2.	Component Trace View	188
E.	CHAPTER SUMMARY.....	192
VIII.	VALIDATION.....	193
A.	APPROACH TO VALIDATION	193
1.	Definitions.....	193
a.	<i>Theoretical Feasibility</i>	193
b.	<i>Interoperability</i>	194
c.	<i>Interoperability Improvement</i>	195
2.	Experimental Design.....	196
a.	<i>Overview</i>	196
b.	<i>Static Group Comparison</i>	196

3.	Sizing the Dissertation Investigation and Risk Management	197
a.	<i>Limited Number of Software Development Tools Analyzed</i>	197
b.	<i>Superficial Exploration for Counter-examples</i>	198
c.	<i>Partial Implementation of Example Tools</i>	198
d.	<i>No Validation against a real-world system</i>	198
e.	<i>Early Implementation</i>	198
B.	CONDUCT OF THE EXPERIMENT	199
1.	“Hello World” Toy Software Scenario	199
2.	“CARA Infusion Pump” Software Scenario	205
C.	RESULTS AND CONFIRMING EVIDENCE OF THE HYPOTHESIS.....	208
1.	“Hello World” Results.....	208
a.	<i>Questions Posed</i>	208
b.	<i>Non-HFSE Results: Hello World</i>	209
c.	<i>HFSE Results: Hello World</i>	209
2.	“CARA Infusion Pump” Software Scenario	216
a.	<i>Questions Posed</i>	216
b.	<i>Non-HFSE Results: CARA</i>	217
c.	<i>HFSE Results: CARA</i>	217
3.	Evidence Confirming the Dissertation Hypothesis	223
D.	INTERNAL AND EXTERNAL VALIDITY AND EXPERIMENTAL IMPLICATIONS	224
1.	Internal and External Validity	224
2.	Sources of Internal Invalidity	224
a.	<i>History</i>	224
b.	<i>Maturation</i>	225
c.	<i>Testing</i>	225
d.	<i>Instrumentation</i>	226
e.	<i>Statistical Regression</i>	226
f.	<i>Selection Biases</i>	227
g.	<i>Experimental Mortality</i>	227
h.	<i>Selection-Maturation Interaction</i>	228
3.	Sources of External Invalidity	228
a.	<i>Interaction of Testing and X</i>	228
b.	<i>Interaction of Selection and X</i>	228
c.	<i>Reactive Arrangements</i>	229
d.	<i>Multiple-X Interference</i>	229
4.	Summary of Experimental Validity	230
E.	CHAPTER SUMMARY	232
IX.	CONCLUSIONS	233
A.	REVIEW OF THE DISSERTATION CONTRIBUTIONS.....	233
1.	Accomplishment of the Research Goal	233
a.	<i>Construction of the HFSE is Feasible</i>	233
b.	<i>HFSE Artifacts Can be Described Mathematically</i>	233
c.	<i>The HFSE Increases Software Tool Interoperability</i>	233
2.	Other Original and Unique Contributions	233

a.	<i>Development of a Software Development Tool Ontology Construction Methodology</i>	<i>233</i>
b.	<i>Construct a Pilot Software Development Tool Ontology.....</i>	<i>234</i>
c.	<i>Use the QFD Methodology to Deploy Software Dependencies other than Quality.....</i>	<i>234</i>
d.	<i>Apply OOMI to the Software Development Tool Domain...</i>	<i>234</i>
e.	<i>Use the HFSE to Provide Perspective Views of the Development Effort.....</i>	<i>234</i>
2.	Potential Long-Term Benefits to the Field of Software Engineering.....	235
a.	<i>Improved Software Development Processes.....</i>	<i>235</i>
b.	<i>Improved Software Products</i>	<i>235</i>
c.	<i>Recognition of Unrealized Software Development Dependencies.....</i>	<i>235</i>
B.	RESEARCH ISSUES ADDRESSED	235
1.	Software QFD.....	236
a.	<i>Questions.....</i>	<i>236</i>
b.	<i>Answer.....</i>	<i>236</i>
2.	Automation of Requirement Prioritization	236
a.	<i>Question.....</i>	<i>236</i>
b.	<i>Answer.....</i>	<i>236</i>
3.	Automation of Dependency.....	236
a.	<i>Question.....</i>	<i>236</i>
b.	<i>Answer.....</i>	<i>237</i>
4.	QFD Dependencies.....	237
a.	<i>Question.....</i>	<i>237</i>
b.	<i>Answer.....</i>	<i>237</i>
5.	QFD and the RH Model	238
a.	<i>Questions.....</i>	<i>238</i>
b.	<i>Answer.....</i>	<i>238</i>
6.	Monitoring Artifacts.....	238
a.	<i>Questions.....</i>	<i>238</i>
b.	<i>Answer.....</i>	<i>239</i>
7.	HFSE Communications.....	239
a.	<i>Question.....</i>	<i>239</i>
b.	<i>Answer.....</i>	<i>239</i>
8.	HFSE and APIs	239
a.	<i>Questions.....</i>	<i>239</i>
b.	<i>Answer.....</i>	<i>239</i>
9.	Missing and Ambiguous Data.....	240
a.	<i>Question.....</i>	<i>240</i>
b.	<i>Answer.....</i>	<i>240</i>
10.	HFSE Extensibility.....	240
a.	<i>Questions.....</i>	<i>240</i>
b.	<i>Answer.....</i>	<i>240</i>
11.	Process Dependencies and the HFSE	240

	a.	<i>Questions</i>	240
	b.	<i>Answer</i>	240
12.		The HFSE and GUI Consoles	241
	a.	<i>Questions</i>	241
	b.	<i>Answer</i>	241
C.		RECOMMENDATIONS FOR FUTURE RESEARCH	241
	1.	Follow-on Hypotheses	242
	2.	Comprehensive Model Validation	242
	3.	Additional Future Research Issues	243
	a.	<i>HFSE Providing a Common Tool View</i>	243
	b.	<i>Tool Replacement in the HFSE</i>	243
	c.	<i>Tool Data Semantics</i>	243
	d.	<i>Specification Tradeoff Elasticity</i>	244
	e.	<i>HFSE Data Representation</i>	244
	f.	<i>Interoperability Tradeoffs</i>	244
	g.	<i>Data Standards and the HFSE</i>	244
	h.	<i>Dependency Paths and Constraints</i>	244
	i.	<i>Method Tailoring</i>	245
	j.	<i>Scalability of the HFSE Approach</i>	245
	k.	<i>Sensitivity Analysis</i>	245
D.		CONCLUDING REMARKS	245
		GLOSSARY	247
		APPENDIX A: CASES USE CASES	261
	A.	INTRODUCTION	261
	B.	CASES TOP-LEVEL USE CASES	262
	C.	USE CASE 1.0: IMPORT TRANSLATORS FROM BABEL	263
	D.	USE CASE 2.0: SOFTWARE ENGINEER SPECIFIES SOFTWARE PROCESS	263
		1. Use Case 2.1: Load Existing Software Process (2 Scenarios)	264
		2. Use Case 2.2: Create Components and Component Attributes	265
		3. Use Case 2.3: Edit Components and Component Attributes (2 Scenarios)	266
		4. Use Case 2.4: Create Steps and Step Attributes	267
		5. Use Case 2.5: Edit Steps and Step Attributes (2 Scenarios)	268
		6. Use Case 2.6: Move (Rearrange) Components	269
		7. Use Case 2.7: Delete Components and Steps (2 Scenarios)	270
		8. Use Case 2.8: Decompose Components into Subcomponents	271
		9. Use Case 2.9: Decompose Steps	272
	E.	USE CASE 3.0: SOFTWARE ENGINEER SPECIFIES COMPONENT DEPENDENCIES	273
		1. Use Case 3.1: Create Dependency	274
		2. Use Case 3.2: Establish Dependency Linkages Between Components (2 Scenarios)	275
		3. Use Case 3.3: Deploy Dependency Through the Development Effort (2 Scenarios)	277

F.	REGISTER COMPONENTS AND STEPS TO EXTERNAL TOOL ARTIFACTS AND ACTIVITIES	277
1.	Use Case 4.1: Map Components to Tool Objects	278
2.	Use Case 4.2: Map Steps to Tool Activities/Methods.....	278
3.	Use Case 4.3: Insert Translators.....	279
G.	USE CASE 5.0: CASES THROUGH MIDDLEWARE MECHANISM COLLECTS/TRACKS ARTIFACTS AND ACTIVITIES.....	279
H.	USE CASE 6.0: SELECT VIEWS OF ARTIFACT DEPENDENCIES	280
1.	Software Engineer Selects a View Based on Single Dependency Link (2 Scenarios)	280
2.	Use Case 6.2: Software Engineer Selects a View Based on Single Component	282
3.	Use Case 6.3: Software Engineer Selects a View Based on a Threshold Dependency Value for a Particular Type of Dependency.....	283
APPENDIX B: CARA INFUSION PUMP REQUIREMENTS FROM REQUISITE PRO		285
A.	INTRODUCTION.....	285
B.	CARA REQUIREMENTS SPECIFIED IN REQUISITE@PRO.....	285
APPENDIX C: SEATOOLS CARA MODEL DESCRIPTIONS		311
A.	INTRODUCTION.....	311
B.	VERSION 1	311
1.	Parent Vertex: Puett_Liang_CARA	311
2.	Parent Vertex: CARA.....	312
3.	Parent Vertex: Management_Module.....	313
4.	Parent Vertex: Pump_Control_Module	314
5.	Parent Vertex: IO_Module	315
B.	VERSION 2	315
1.	Parent Vertex: Puett_Liang_CARA	316
2.	Parent Vertex: Infusion_Pump.....	317
3.	Parent Vertex: CARA.....	318
4.	Parent Vertex: Management_Module.....	319
5.	Parent Vertex: Pump_Control_Module	320
6.	Parent Vertex: IO_Module	321
7.	Parent Vertex: Line_Monitor	322
8.	Parent Vertex: Resuscitation_File.....	323
9.	Parent Vertex: Module1	324
10.	Parent Vertex: Voting_Element	325
11.	Parent Vertex: Alarm_Controller1	326
12.	Parent Vertex: Display_Driver	327
13.	Parent Vertex: Display	328
14.	Parent Vertex: AirOK_Monitor	329
15.	Parent Vertex: EMF_calculator	330
16.	Parent Vertex: Impedance_Monitor	331
17.	Parent Vertex: BP_Calculator.....	332

LIST OF REFERENCES	333
INITIAL DISTRIBUTION LIST	343

LIST OF FIGURES

Figure 1	Life-Time Costs of Software Development.....	1
Figure 2	Typical Software Development Process Interaction.....	4
Figure 3	Holistic Model of Software Process Interaction	15
Figure 4	Software Evolution as a Feedback and Control System	27
Figure 5	Directed Hypergraph with Incidence Matrix	33
Figure 6	Software Evolution Processes with CASES (from [HARN99c]).....	34
Figure 7	QFD Enhances Cross-Functional Communication (after [COHE95])	37
Figure 8	The Quality Continuum (after [ZULT92, 93])	42
Figure 9	Kano diagram for Requirements (after [ZULT90]).....	43
Figure 10	First Level QFD Matrix -- The "House of Quality"	46
Figure 11	Example of a Simplistic QFD Matrix Deployment	48
Figure 12	Simplistic SQFD Model drawn as a Process Diagram	49
Figure 13	SQFD -- Deployment of the "Customer's Voice" (after [ZULT90])	52
Figure 14	SQFD -- Matrix Deployment as a Process Diagram.....	53
Figure 15	SQFD for Embedded Systems (after [THAC90]).....	54
Figure 16	Embedded System SQFD Displayed as a Development Process Model.....	54
Figure 17	SQFD and Concurrent Engineering (after [THAC90])	55
Figure 18	Typical QFD Symbols for Degree of Relationship.....	56
Figure 19	3D View of Matrix Interaction (after [DEAN92]).....	58
Figure 20	Federation Interoperability Object Model (from [YOUN02a])	61
Figure 21	Middleware Based Translation Using the FIOM (from [YOUN02b])	62
Figure 22	Role of Ontology in FIOM Construction (from [YOUN02b])	68
Figure 23	Protégé Screen Shot	72
Figure 24	Synchronization of Perspectives in the Rational Process (from [KRUC96])..	75
Figure 25	Intellectual Activity in the Rational Process (from [RATI03])	76
Figure 26	Rational Unified Process (from [KRUC96])	80
Figure 27	SWEBOK Software Requirements Taxonomy (after [SWEB01]).....	87
Figure 28	SWEBOK Software Tool Taxonomy (after [SWEB01]).....	88
Figure 29	Feature Model of the PSDL Timing Constraints of SEATools (after [HASN03]).....	97
Figure 30	Construction of an Affinity Diagram.....	99
Figure 31	Ontology Inter-relationship (after [HASN03])	100
Figure 32	Excerpt of the Requisite®Pro Feature Model (after [HASN03])	103
Figure 33	Excerpt from the SEATools Feature Model (after [HASN03]).....	108
Figure 34	Software Development Tool Federation Ontology (after [HASN03])	115
Figure 35	Excerpt of the Class Structure of the Requisite®Pro Ontology (after [HASN03]).....	116
Figure 36	Class Structure of the SEATools Ontology (after [HASN03]).....	117
Figure 37	Communication Class Inter-relationships (from [HASN03]).....	118
Figure 38	Sample Relational Hypergraph (from [HARN99c]).....	123
Figure 39	Deployment of Risk Example.....	128

Figure 40	Hypergraph Representation of the QFD Example	129
Figure 41	Weighted Digraph Representation of the QFD Example	129
Figure 42	Hypergraph with Risk Dependency Values	139
Figure 43	Subgraphs Trimmed with Dependency Threshold	140
Figure 44	Weighted Digraph Example.....	140
Figure 45	Component Trace from A3 with Threshold 2.....	141
Figure 46	Component Trace from A3 with Threshold 8.....	141
Figure 47	AHP Clustering Example.....	150
Figure 48	FIOM Construction Methodology	155
Figure 49	Protégé Base XML Schema	157
Figure 50	Software Development Tool XML Schema Excerpt.....	158
Figure 51	SEATools XML Schema Excerpt.....	159
Figure 52	Requisite®Pro XML Schema Excerpt.....	160
Figure 53	Requirement Federation Entity (FE).....	161
Figure 54	Rational Rose Extensibility Interface (from [ROSE02])	163
Figure 55	Insertion-Sort Algorithm Pseudo-Code (from [CORM91])	164
Figure 56	Merge-Sort Algorithm Pseudo-Code (from [CORM91])	164
Figure 57	CASESv1.1 New Project Screenshot.....	176
Figure 58	CASESv1.1 Project Schema Creation Dialog	177
Figure 59	CASESv2 Project Schema Creation Process	178
Figure 60	CASESv2 Completed Project Schema.....	179
Figure 61	IBIS Evolutionary Process Model in CASESv2	180
Figure 62	CASESv2 Dependency Creation Dialog	181
Figure 63	C4I Systems Requirements Analysis Step (after [HARN99c])	183
Figure 64	CASESv2 Data Integration via Import CSV File (Requirements variant 2 version 3)	184
Figure 65	QFD Matrix: Requirements x Specifications (Dependency: Reqt Risk)	185
Figure 66	User-Defined Example from Chapter IV	187
Figure 67	QFD Matrix: R x S (Dependency: Risk).....	187
Figure 68	User-Defined Views with Threshold = μ and Threshold = $\mu + 1\sigma$	188
Figure 69	Component Trace Example from Chapter IV	189
Figure 70	QFD Matrices for Component Trace Example.....	190
Figure 71	QFD Trace from A3 (Upstream & Downstream) Threshold 2.....	190
Figure 72	QFD Trace from A3 (Upstream) Threshold 8	191
Figure 73	Hello World Development Process.....	200
Figure 74	SEATools Hello World Prototype Variants 1 & 2, Versions 1 & 2	202
Figure 75	Hello World Implementation Variants 1 & 2, Versions 1 & 2	203
Figure 76	SEATools Hello World Prototype Variant 2, Version 3.....	204
Figure 77	Hello World Implementation Variant 2, Versions 3	205
Figure 78	CARA Software Development Process	207
Figure 79	Hello World Project Schema	210
Figure 80	Component Trace: R2.2-2.3, $t = 1$	211
Figure 81	QFD Matrix: S1.1 x C1.1, $d = \text{Risk}$	211
Figure 82	Component Trace: S1.1-O1.2, $t = 1$	212
Figure 83	Dependency Threshold: S1.1 x C1.1, $d = \text{Difficulty}$, $t = \mu$	213
Figure 84	Component Trace: F2.2-1, $t = 3$	214

Figure 85	Dependency Threshold: S2.3 x C2.3, $d = \text{AHP Priority}$, $t = \mu + 0.5\sigma$	215
Figure 86	QFD Matrix: R2.3 x S2.3, $d = \text{Priority}$	216
Figure 87	Dependency Threshold: R1.1 x S1.1, $d = \text{Safety}$, $t = \mu + 1.5\sigma$	218
Figure 88	Dependency Threshold: R1.1 x S1.1, $d = \text{AHP Priority}$, $t = \mu + 2.5\sigma$	219
Figure 89	Dependency Threshold: R1.1 x S1.1, $d = \text{Risk}$, $t = \mu + 1.0\sigma$	220
Figure 90	Component Trace: R1.1-34, $t = 3$	221
Figure 91	Component Trace: S1.1-O1.2.3.5, $t = 1$	222
Figure 92	CASES Context Diagram	261
Figure 93	CASES Top-Level Use Cases.....	262
Figure 94	Use Case 1.0: Import Translators from Babel	263
Figure 95	Use Case 2.0: Software Engineer Specifies Software Process	264
Figure 96	Use Case 3.0: Software Engineer Specifies Component Dependencies.....	274
Figure 97	Use Case 4.0: Register Components and Steps to External Tool Artifacts and Activities	277
Figure 98	Use Case 5.0: Register	279
Figure 99	Use Case 6.0: Select Views of Artifact Dependencies	280
Figure 100	Top Level CARA Model v1	312
Figure 101	CARA Software Model v1.....	312
Figure 102	Management Module v1	313
Figure 103	Pump_Control Module v1.....	314
Figure 104	IO_Module v1	315
Figure 105	CARA Software Model v2.....	316
Figure 106	Infusion_Pump Pin-outs v2.....	317
Figure 107	CARA System Modules v2.....	318
Figure 108	Management_Module v2	319
Figure 109	Pump_Control_Module v2.....	320
Figure 110	IO_Module v2.....	321
Figure 111	Line_Monitor v2	322
Figure 112	Resuscitation_File v2.....	323
Figure 113	Module1 v2	324
Figure 114	Voting_Element v2	325
Figure 115	Alarm_Controller1 v2.....	326
Figure 116	Display_Driver v2.....	327
Figure 117	Display v2	328
Figure 118	AirOK_Monitor v2	329
Figure 119	EMF_Calculator v2.....	330
Figure 120	Impedance_Monitor v2.....	331
Figure 121	BP_Calculator v2.....	332

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1	Lehman's Laws of Software Evolution (after [LEHM97]).....	30
Table 2	Akao "Matrix of Matrices" Summary (after [COHE95])	51
Table 3	Overview of the Enterprise Ontology (after [USCH98]).....	67
Table 4	DAML Ontology Library: Software Tool Ontology (after [DAML03])	89
Table 5	DAML Ontology Library: Software Engineering Ontology (after [DAML03]).....	90
Table 6	QFD Matrices for Performing Object-Oriented Analysis (after [LAMI95])...	91
Table 7	Requisite®Pro Feature List (after [HASN03]).....	107
Table 8	SEATools Feature List (after [HASN03])	112
Table 9	Common Characteristics for Software Development Tool Federation (after [HASN03]).....	114
Table 10	"Downstream" Dependency Deployment	125
Table 11	"Upstream" Dependency Deployment.....	127
Table 12	QFD Matrix for Risk Deployment Example.....	128
Table 13	Dependency Thinning (Dependency = 10).....	132
Table 14	Dependency Concentration (Dependency = 40).....	132
Table 15	QFD Coverage Analysis Example.....	136
Table 16	Coverage Analysis Example: Side-by-Side Comparison	137
Table 17	QFD Coverage Analysis Example.....	137
Table 18	Coverage Analysis Example: Side-by-Side Comparison	138
Table 19	Example Absolute Importance Scale	144
Table 20	Absolute Importance Valuation Scheme	145
Table 21	Relative Importance Valuation Scheme.....	145
Table 22	Ordinal Importance Valuation Scheme.....	146
Table 23	Example AHP Comparison Matrix (after [SAAT80]).....	148
Table 24	AHP Comparison Valuation Scheme.....	148
Table 25	AHP Example Normalized Priority Values	149
Table 26	Excerpt from CARA Infusion Pump Requirements	150
Table 27	Large Granularity Pseudo-Code CSV File	164
Table 28	Fine Granularity Pseudo-Code CSV File (after [CORM91])	165
Table 29	Customer Requirements Variant 1 Version 1 (R1.1).....	201
Table 30	Customer Requirements Variant 2 Version 2 (R2.2).....	201
Table 31	Developer-Customer Question & Answer Version 1 & 2 (Q1.1, Q2.2).....	201
Table 32	Software Specifications Variants 1 & 2 Versions 1 & 2 (S1.1, S2.2)	202
Table 33	Code Variant 1 Version 1 (C1.1)	202
Table 34	Code Variant 2 Version 2 (C2.2)	203
Table 35	Customer Feedback Version 1 & 2 (F1.1, F2.2).....	203
Table 36	Customer Requirements Variant 2 Version 3 (R2.3).....	204
Table 37	Developer-Customer Question & Answer Version 3 (Q2.3).....	204
Table 38	Software Specifications Variant 2 Version 3 (S2.3).....	204
Table 39	Code Variant 2 Version 3 (C2.3)	205
Table 40	Customer Feedback Version 3 (F2.3).....	205

Table 41	Excerpt of CARA Questions and Answers (after [WRAI01b]).....	206
Table 42	Excerpt from CARA Model 1, Specifications 1.1	208
Table 43	Most Safety Critical Components: $R \times S, d = \text{Safety}, t = \mu + 1.5\sigma$	218
Table 44	Most Important Components: $R \times S, d = \text{AHP Priority}, t = \mu + 2.0\sigma$	219
Table 45	Most Risky Components: $R \times S, d = \text{Risk}, t = \mu + 1.0\sigma$	220
Table 46	Component Trace: R1.1-34, $t = 1$	221
Table 47	Component Trace: S1.1-O1.2.3.5, O1.2.3.5.9, E38, $t = 1$	222
Table 48	Summary of Sources of Invalidity	231
Table 49	Actor-System Responses for Use Case 1.0.....	263
Table 50	Actor-System Responses for Use Case 2.1 (Scenario 1)	264
Table 51	Actor-System Responses for Use Case 2.1 (Scenario 2)	265
Table 52	Actor-System Responses for Use Case 2.2.....	265
Table 53	Actor-System Responses for Use Case 2.3 (Scenario 1)	266
Table 54	Actor-System Responses for Use Case 2.3 (Scenario 2).....	267
Table 55	Actor-System Responses for Use Case 2.4.....	268
Table 56	Actor-System Responses for Use Case 2.5 (Scenario 1)	268
Table 57	Actor-System Responses for Use Case 2.5 (Scenario 2).....	269
Table 58	Actor-System Responses for Use Case 2.6.....	269
Table 59	Actor-System Responses for Use Case 2.7 (Scenario 1)	270
Table 60	Actor-System Responses for Use Case 2.7 (Scenario 2).....	271
Table 61	Actor-System Responses for Use Case 2.8.....	272
Table 62	Actor-System Responses for Use Case 2.9.....	273
Table 63	Actor-System Responses for Use Case 3.1	274
Table 64	Actor-System Responses for Use Case 3.2 (Scenario 1)	275
Table 65	Actor-System Responses for Use Case 3.2 (Scenario 2)	276
Table 66	Actor-System Responses for Use Case 3.3 (Scenario 1)	277
Table 67	Actor-System Responses for Use Case 3.3 (Scenario 2).....	277
Table 68	Actor-System Responses for Use Case 4.1	278
Table 69	Actor-System Responses for Use Case 5.0.....	279
Table 70	Actor-System Responses for Use Case 6.1 (Scenario 1)	281
Table 71	Actor-System Responses for Use Case 6.1 (Scenario 2)	282
Table 72	Actor-System Responses for Use Case 6.2.....	283
Table 73	Actor-System Responses for Use Case 6.3.....	284

ACKNOWLEDGMENTS

It would have been impossible for me to undertake an endeavor of this size and scope without the assistance of numerous individuals. It is with sincerest thanks that I would like to acknowledge their help, contribution, and guidance to my education and this accomplishment.

First, I would like to acknowledge the Masters students with whom I worked: LT Tolga Demirtas, Turkish Navy; LT Neji Hasni, Tunisian Navy; and MAJ Art Clomera, U.S. Army. Each of them challenged me academically as much as I challenged them and without their help it would have been impossible for me to tie together the many threads of this research.

I would like to thank Professor Richard Reihle, who even though not a part of my dissertation committee, did as much for teaching me what it means to be a software engineer as anyone I have met.

Next, I would like to acknowledge the members of my dissertation committee: Professors Rasmussen, Michael, Shing, and Ludlow. Their tireless efforts to steer me in the right directions, clarify my understanding of complex material, and offer suggestions of how to improve the dissertation were invaluable. Professor Luqi, as my dissertation supervisor, nudged me when I needed nudging and provided me the guidance I needed to get the research completed.

Finally, I would like to thank my wife, Peg, and sons, Michael and Gregory who have faithfully supported me in my every enterprise.

To each and every one of you -- Thank-you.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION

The Department of Defense and the nation become more reliant on software every day; yet, consistently producing high quality software on-time and within budget that fully meets customers' requirements is challenging for many reasons. While there have been many investigations to address this challenge, "understandability" (or the lack thereof) is considered to be a major issue. Exacerbating this issue are the needs for holistic and coherent development processes and adequate requirements engineering integration.

1. Software Understandability

Consider the following charts (see Figure 1) related to total costs during the Software Development Lifecycle and the cost impact of maintenance -- where "maintenance" refers to all activities beyond the initial release.

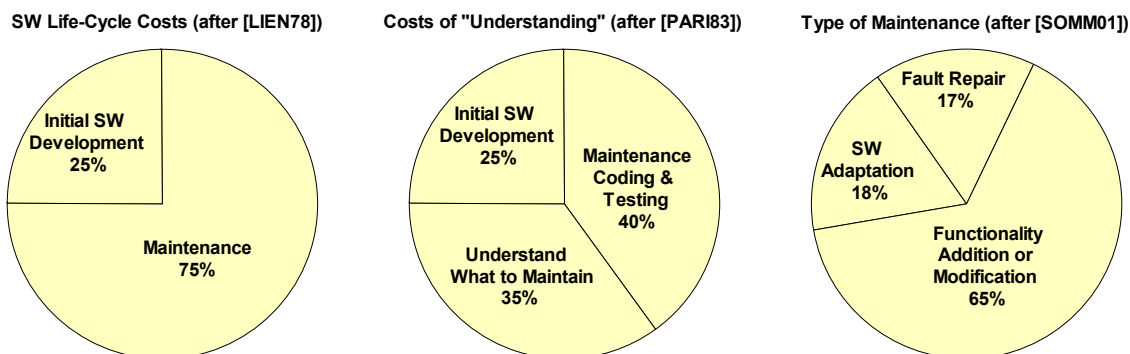


Figure 1 Life-Time Costs of Software Development

The first pie chart in Figure 1 illustrates that maintenance costs dominate initial software development costs by a factor of three. The second chart illustrates that costs associated with "understanding" exactly what to maintain account for almost half of the maintenance costs (i.e., studying and understanding what to add, correct, or change is almost as expensive as actually performing maintenance coding and testing). While data supporting the first two charts is somewhat dated, [MEYE97] and [BOEH95] confirm that more recent evidence still supports these older observations. Finally, the third pie chart illustrates how the maintenance costs are divided; illustrating that "understanding"

is required in three different contexts depending on the particular type of maintenance to be performed. Together, these charts imply that it is not enough to simply improve individual tools and/or the process used to develop software; software engineering researchers must take a more holistic view of how software is handled over its entire lifecycle and improve how the tools and processes work together over long periods of time to overcome the understandability challenge. This holistic view considers the whole of the software lifecycle and does not simply focus on a single iteration of the lifecycle, or a single process within an iteration, or a single set of software artifacts produced by an individual process. Unfortunately, rather than taking such a holistic approach, the lion's share of software engineering research has been devoted to meeting this challenge by improving specific aspects of the software development process (e.g., requirements engineering, reuse, testing), by improving the software development process itself (e.g., evolutionary prototyping, spiral development), or by improving individual tools for these processes (e.g., Rational Rose, Requisite®Pro, DOORS). Although "Software Evolution" has become a mainstream software engineering subfield, there has still been little unifying research that attempts to determine the best way that these tools and models should (and could) interact to address the "understandability" problem directly.

There appear to be two aspects to this problem of understandability. First, the engineer must always have an "understanding" of what the customer wants (even years after the original requirements were articulated). Second, the engineer needs to have an "understanding" of what previous software engineers have done to address the customer's requirements. While documentation is good at recording decisions and agreements, it is ineffective at recording the detailed knowledge required to understand how a software program really works [MCBR02]. There are at least two approaches that could be taken to solve the understandability problem.

a. Produce Tools to Promote Understandability of Customer Desires and Legacy Software Development Efforts

One approach would be to build some tools: a tool that does a better job of requirements engineering by continually (over the lifecycle of development) querying customers as to what they want and how those desires impact all artifacts of the development, and a second tool which could intelligently parse through pertinent current

and past software development artifacts recognizing and extracting just the information the engineer needs at exactly the time it is needed (and it has to be able to do this with any artifact, produced by any tool).

b. Produce a Framework that Promotes Understandability of Customer Desires and Legacy Software Development Efforts

Another approach would be to establish a holistic framework where all the information is recorded as it is created and the relationship of the artifact (to all other artifacts) is automatically tracked throughout the lifecycle. The framework would be flexible enough to work with any development tool or model. The relationship of a new piece of information to all others is established on creation and the software engineer uses the dependencies of the artifacts to extract relevant information (using engineer defined contexts or views). However, just recording the information is not enough to justify the additional overhead of entering such information into a system. As [LEHM69], a pioneer in the field of software evolution, stated, "The manager faced with the daily problems of meeting a deadline will always first abandon methodology and systematics." Engineers will not use such a framework unless it can help instantiate detailed consequences of explicit high-level decisions and help to propagate consequences of changes.

Of these two approaches for tackling the "understandability problem," the "tools" solution appears to be beyond current technological capability; however, the "framework" option shows promise and is the focus of this dissertation research.

2. Holistic Development

Software engineering research is typified by developing or improving individual aspects of software development. Examples include research into software evolution models, requirements engineering, risk and cost estimation, software reuse, prototyping, testing, software integration, software maintenance, re-engineering, performance analysis, domain analysis, architecture design, etc. These individual aspects of software development necessitate that the software engineer provide any needed interface between different development models and tools (see Figure 2).

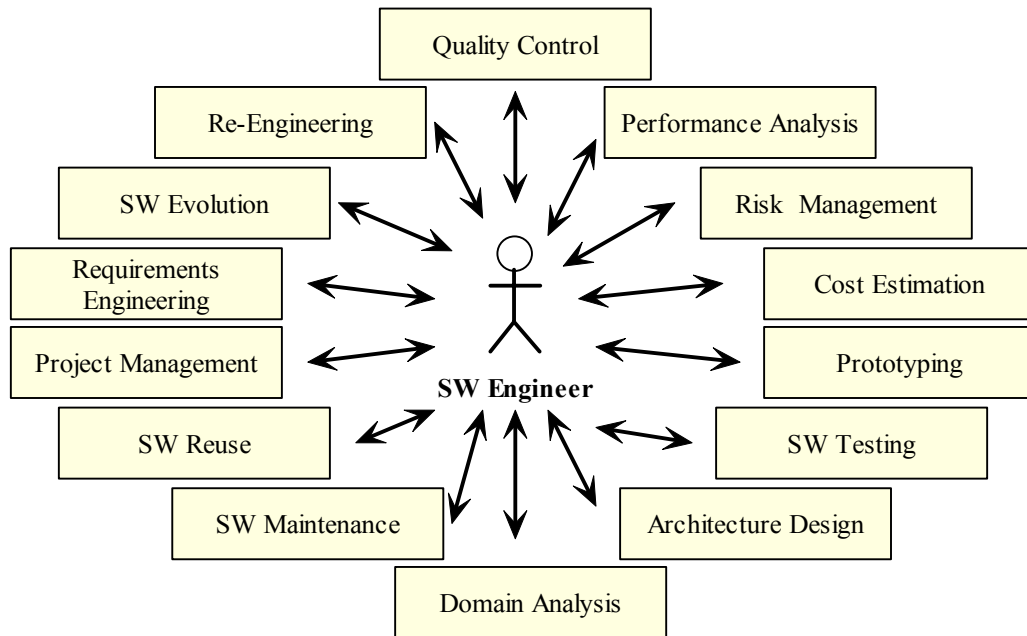


Figure 2 Typical Software Development Process Interaction

While there has been plenty of research into the development of Integrated Software Development Environments (ISDEs) [LEHM87], [BROW92, 93], [KADI92a, b], [ARCA95] and software development tool suites [RATI98], [KRUC96], there has been little research into holistic models to define how these various threads and processes could (and should) most efficiently and effectively interact over the entire lifecycle of the software development effort. Currently, there is inadequate long-term communication of risk and requirements across disjoint tools and models.

A proponent of agile software development processes, [MCBR02] notes that while there has been a great deal of research effort and money devoted to trying to remove the human from the software development process, this effort has met with little success. Software engineers can only automate parts of the process; they cannot automate processes that require rich interactions between people. For instance, the gap between requirements specification and design specification cannot be filled with automated tools; only skilled developers can bridge such a large and complex gap. [MCBR02] states that most parts of the software development process that can be automated have been automated; what remains is to make the best use of the tools at hand. As engineers automate successive parts of the development process, they are not

able to eliminate complexity; they are only able to manage it. Thus, there is much to be gained by improving the way in which developers use existing tools, rather than constantly developing new ones.

The development of a holistic framework potentially provides seamless interoperability between software development processes allowing software systems to be produced more efficiently and reliably with high quality. Additionally, the existence of such a framework enhances the discovery of dependencies among different aspects of the software engineering processes. The hope is that it will enable software engineers to discover process improvements. The long-term goal of this research is to support all aspects of software engineering; however, the immediate goal presented in this dissertation is to demonstrate the theoretical feasibility of integrating a selected subset of models and tools using a holistic framework.

3. Requirements Engineering

At least one third of software development projects (including military projects) run into trouble for reasons that are directly related to requirements gathering, requirements documenting, and requirements management. A Standish Group (1994) study (as reported in [LEFF00]) noted that the three most commonly cited factors that caused software projects to be "challenged" were (with some overlap in percentages) as follows:

- Lack of user input: 13% of all projects,
- Incomplete requirements and specifications: 12% of all projects, and
- Changing requirements and specifications: 12% of all projects.

Amplifying this problem is the fact that fixing errors missed during the requirements phase can be extremely costly. Depending on how much later the error is discovered, the party acquiring a software system may incur costs for re-specification, redesign, change orders, recall of defective versions, service costs, documentation, and retraining. The Federal Aviation Administration's (FAA's) Advanced Automation System (AAS) is an example of how significantly costs rise when requirements errors are identified and addressed at later stages in a development effort [GAOT98]. The data related to requirements management demonstrates the following two things:

- Requirements errors are the most likely category of error.
- Requirements errors tend to be the most expensive error to fix.

A requirements-based methodology used widely in the global product industry to meet these same challenges of understandability, quality, safety, and reliability in highly constrained time/budget development environments is a methodology known as "Quality Function Deployment" or "QFD". QFD is focused on meeting three major challenges in the development of products: ensuring that the "voice" of the customer is adequately transferred to each segment of the development effort, ensuring that there is no loss of development information, and ensuring that different segments of the development effort are working in concert to satisfy the same set of customer requirements. As stated by one of the pioneers of QFD in the United States [CLAU88], QFD "facilitates a holistic response to customer needs." QFD allows engineers to summarize basic data in usable form. It allows marketing executives to directly hear the customer's desires. It allows general managers to discover strategic opportunities. QFD encourages all of the different groups in a development effort to work together to understand one another's priorities and goals [HAUS88].

However, the use of QFD as applied to software development has been very limited. In a 1995 survey of thirty-seven major software vendors (companies that produce and sell software as a major component of their business operations) only 16% of them used Software QFD (SQFD) [HAAG96]. There are two reasons cited for this:

- SQFD has not been rigorously integrated throughout existing software development models and tools.
- Material published to date about SQFD has been highly conceptual with little pragmatic application. Companies that are successfully using SQFD are reluctant to offer up their practical experience and the competitive advantage that the SQFD has afforded them.

As cited in the survey, all major software vendors perceived that the requirements-gathering tools in their respective software development methodologies were not adequate. QFD is a methodology that specifically targets the problems in requirements management by ensuring that the "voice of the customer" is appropriately deployed throughout the follow-on phases of the design and is not forgotten or misinterpreted immediately after the requirements capture phase. However, as of now,

QFD must be manually integrated into software development processes; there are no mechanisms for having it automatically interact with existing software development tools. This research seeks an improvement to this status quo by incorporating automated QFD mechanisms into a software evolution model.

4. Coherent Development

In most software development efforts the requirements engineers can readily point out which requirements are the best analyzed and most clearly stated. The software architects proudly show off portions of the architecture that employ the best technology and will be encapsulated in the most cohesive and least coupled modules. The programmers can point to the code that is most elegantly coded and the testers can exactly identify the portions of code that have been most thoroughly tested and likely have the fewest defects. Unfortunately, it is only by accident that any of these "best" areas of development will coincide with the particular functionality that the customer thinks provides him the greatest value -- "mediocre software is the result" [ZULT93]. Such observations are not new. [LEHM69] in commenting on how hierarchical management attempts (but fails) to handle this problem states:

The consequences of this procedure [hierarchical project management] are apparent. Communications within a group, and more importantly, between different groups, tend to be random and a matter of chance. Personal relationships between individuals exert a strong influence on final system structure, distribution and content. Optimisation, if any, is local within each group. Thus the system becomes an assembly of its parts, amorphous, redundant and with random, largely invisible, communication. Attempts to debug, improve or enlarge the system become very difficult tending to cause its collapse.

Thus, while the problems of incoherence in software development were recognized as far back as 1969, little has changed to correct the problem.

It has been noted that traditional, non-software product development can also be incoherent with respect to the attributes that customers associate with a quality product. However, the use of QFD addresses this problem by focusing all of the company's development efforts in the same direction [CLAU88]:

Although much of the initial attention given to QFD in the U.S. focused on its formats, QFD's style of organizational behavior is even more

important. This style emphasizes multifunctional teams that work to achieve consensus about customer requirements and product-expectation requirements. This helps break down segmentation among the various corporate functions and brings the collective wisdom of the corporation to bear on the product. The team produces product specifications responsive to customer needs that will be vigorously worked on by all the functions. This compares with today's style of each function doing its own thing and then throwing the result over the wall to the next function... There's a tendency for specialists to stay cloistered within their specialties. Individually, they contribute tremendous specialized knowledge, but there's some difficulty in integrating that knowledge so that it provides a holistic response to customer needs.

[ZULT92] reinforces this view for software:

Traditional [software] development is unfocused with respect to quality. The best analyzed requirements are not the best designed. The best designed elements are not the best coded. The best code is not the best documented. It is only by chance that the best efforts of one phase receive the best efforts of a subsequent phase... Incoherent processes are inefficient and expensive ways to satisfy customers, often forcing trade-offs of quality for schedule or resources.

What is needed is a holistic approach that allows the software developer to have visibility over the more important (as opposed to less important) aspects of the development effort so that they may receive appropriate attention and resources. Thus, by successfully integrating QFD into the software development evolution process, software engineers may later extract particular slices of the development effort that have particular meaning, allowing them to work in a more coordinated and coherent manner, make better decisions, and produce better software.

5. Software Safety

"Software safety" has become a significant system issue. Increasingly, software is used in safety-critical applications during which if the software fails, there can be injury, loss of life, property damage, or efficacy losses. Managing the safety risk associated with the software control of critical functions requires a structured, disciplined system safety management and engineering approach that focuses on the unique aspects of software in a systems context. Holistic approaches provide the software engineer visibility of the dependencies between safety-related software artifacts. Such visibility

permits the engineer to confidently modify safety-critical software, knowing that the dependencies associated with changes are being tracked. Producing safety-critical software is an expensive and time-consuming endeavor. Much of the expense and time is due to identifying all "knock-on" effects when modifying safety-critical parts of the software. A holistic model can make it possible to identify all of these "knock-on" effects in a timely and systematic manner. The model itself can be automated to some extent, freeing the software engineer from the mechanistic aspects of injecting quality into software systems.

Many of these safety-critical problems and requirements are the same as those faced by NASA. QFD has been used successfully as a methodology that manages safety-critical requirements for large complex space systems [DEAN92] by ensuring that safety requirements are deployed throughout the follow-on phases of the design.

6. Solving these Problems with the Holistic Framework For Software Engineering (HFSE)

The following elements are required to deliver any reliable, safe, and quality software system of any significant scope [LEFF00]:

- A pragmatic process for defining and managing the requirements for software;
- A solid, rigorous, and repeatable methodology for the design and development of software;
- The application of various proven, innovative, techniques for verifying and validating that the software is safe and effective; and
- Extraordinary skills and commitment on the part of both the software development and software quality assurance teams.

This dissertation implements these elements by employing a Holistic Framework for Software Engineering (HFSE) that not only lets software engineers work faster, but lets them work smarter with greater understanding of customer desires and previous development work. The Holistic Framework is established by embedding the relevant portions of the Quality Function Deployment methodology into the Relational Hypergraph Computer Aided Software Evolution model, then integrating this extended evolution model with a Federation Interoperability Object Model created from the tools and models used by the development team. While there is no "silver bullet" that will solve all software development problems, the HFSE provides an improved requirements-

based model upon which to develop safe, reliable software, produced on-time and on-budget that fully meets the customers' requirements.

This research constitutes an initial investigation into the development of the HFSE that establishes mechanisms by which existing software development tools and models can work together. This dissertation demonstrates that establishing a mathematical framework that allows existing software engineering process models (and tools supporting those models) to seamlessly interact is technically feasible. Moreover, this dissertation presents the methods and principles needed to realize such a framework and the associated computer aids. The longer-term future goals of this line of research are to actually improve the efficiency of software development processes and to improve previously developed software's quality, safety, and reliability by applying the framework to specific development efforts.

B. RESEARCH HYPOTHESIS AND METHODOLOGY

1. Research Hypothesis

The following is a statement of the Dissertation Hypothesis:

It is theoretically feasible to integrate a selected set of software development tools and/or models through application of a Holistic Framework for Software Engineering (HFSE),

Where

- The HFSE consists of an extended Software Evolution model (extended with Quality Function Deployment (QFD)) integrated with a Federation Interoperability Object Model (FIOM) of the subordinate software development tools/models.
- The integrated tool/model set provides additional interoperability (i.e. additional data exchange and joint task execution) beyond that interoperability available prior to the application of the HFSE to the software development tool set.

2. Research Methodology

This research was completed by executing five major tasks: developing a software development tool ontology, integrating QFD into the Relational Hypergraph model of software evolution, creating an object model of software development tools and

integrating that model with the evolution model, prototyping the HFSE by extending the Computer-Aided Software Evolution System (CASES), and applying the HFSE to a set of software development tools to provide confirming evidence for the research hypothesis.

a. Development of a Software Tool Artifact Ontology

The first step in this research was to identify and define the characteristics of software development process models and tools so that they can be used to properly extend the Relational Hypergraph Software Evolution Model and be used to construct an Object Federation for interoperability. The approach to this portion of the investigation was to analyze the structure, inputs, and outputs of a small set of individual tools: Rational Corporation's Requisite®Pro -- a requirements management tool, and the Software Engineering Automation Tools (SEATools) -- a software development prototyping suite. This included performing a domain analysis (of this subset of tools) and building a feature model of that domain [CZAR00]. Next, the main artifact attributes were considered in the context of the objects needed for establishing an Object Federation. Using this context the characteristics were holistically defined within a software development tool ontology [USCH96]. The bulk of this portion of the dissertation research is presented in Chapter III.

b. Integration of QFD and the Evolution Model

Actually embedding key portions of the QFD methodology within the Relational Hypergraph model required an examination of requirements prioritization, requirements uncertainty, project risk, performance trade-offs between software specifications, and appropriateness of software metrics to measure outcomes. Next, additional objects and attributes were added to the hypergraph model to account for QFD dependencies. The correlation weightings that form the core of QFD were captured as edge weights in the Relational Hypergraph model and priority weightings were established as additional component attributes to be deployed among the software artifacts. The hypergraph requirements model was integrated with the IBIS (Issue Based Information System) evolutionary prototyping model from [IBRA96] which stemmed from the seminal work of [KUNZ70]. This model links the requirements to their rationale, which consists of the positions of various stakeholder groups on the relevant

requirements issues, and is useful for negotiating and resolving differences on requirements issues between different stakeholders. This is a qualitative model. One of the objectives in integrating the hypergraph model with QFD was to provide quantitative models of priorities that are sensitive to dependencies and disagreements among stakeholders. However, one of the weaknesses of QFD is its general treatment of the requirements as independent entities in the mathematical analysis. In reality, requirements are subject to a complex set of interdependencies that are captured by the hypergraph model. By integrating the two models, both aspects (qualitative and quantitative) are now available for decision support. The bulk of the research supporting this portion of the dissertation is presented in Chapter IV.

c. Creation of a Federation Interoperability Object Model

Young presents an Object-Oriented Model for Interoperability (OOMI) of heterogeneous systems [YOUN01, 02a, 02b]. He developed this model for use in establishing interoperability of military C4I systems. However, in this research this same model was applied to a different domain -- the interoperability of software development tools and models. The OOMI relies on the collection of real-world entities used to define the interoperation of a specific collection of systems, which is called a Federation Interoperability Object Model (FIOM). The software development tool ontology provided an object framework from which a partial software development tool FIOM was constructed. The partial FIOM contained relationships between classes, packages, interfaces, and other elements used in the software process models and tools. Next, the FIOM was integrated with the extended evolution model to form an implementation of the HFSE. This portion of the dissertation research is presented in Chapters V and VI.

d. Prototype the HFSE by Extending CASES

The fourth major task in completing this research involved developing a proof of concept computer aid that was later used to demonstrate the software engineering contributions presented in the dissertation. The work here relied on extending a previous software evolution system -- the Computer-Aided Software Evolution System (CASES). The main extensions to CASES involved: providing a graphic editor so that a user could define their own unique software development process, embedding QFD into the evolution system so that artifact dependencies could be defined

and deployed throughout the software development effort, and providing user-selectable views that isolated particular "slices" of the underlying hypergraph -- providing software engineering informative decision support. The extensions to CASES are presented in Chapter VII and use cases for CASES are presented in Appendix A.

e. Application of the HFSE to a Software Development Scenario

The last major task in the dissertation research was to apply the HFSE in a software development scenario to provide confirming evidence for the research hypothesis. Here, Requisite®Pro and SEATools were unified by the HFSE and applied to a particular software scenario. This scenario involved using a specified set of software requirements (specified within Requisite®Pro) for the Computer Aided Resuscitation Algorithm (CARA) software for a casualty intravenous fluid infusion pump. Next, the scenario required using SEATools to construct a software prototype that embodied those requirements. The interoperability benefits provided through the use of the HFSE in this software development scenario were recorded and provide confirming evidence in support of the dissertation hypothesis.

The dissertation experiment is a static group comparison in which a small representative subset of tools/models was used to show that the HFSE can be used to unify them and that the interoperability of the subset of tools was improved. Campbell and Stanley [CAMP63] point out that this comparison is best characterized as a pre-experiment because it falls short of an unbiased application of the scientific method. The results (and shortcomings) of this experiment are presented in Chapter VIII.

C. OVERVIEW OF THE HOLISTIC FRAMEWORK FOR SOFTWARE EVOLUTION

The Holistic Framework for Software Engineering is both a conceptual framework for establishing interoperability between software development tools as well as a methodology (with tool support) that assembles the necessary objects and interoperability constructs. The HFSE was established by embedding the relevant portions of the QFD methodology into the already existing Relational Hypergraph Computer-Aided Software Evolution model and then integrating this extended evolution model with a FIOM created from the tools and models used in software development.

1. Software Evolution

Central to this holistic framework is software evolution. A software evolution system must provide strong version control of all artifacts produced during system development as well as track the dependencies of artifacts. In small projects (which, as a rule of thumb [MCBR02] characterizes as requiring less than 100 man-years of effort) it is less expensive to scan through the application for the impact of a change than it is to slow the project down by insisting on complete requirements traceability. However, for large software engineering projects requiring over 100 man-years of effort, the trade-off is exactly the opposite. Traceability of all requirements must occur because it is prohibitively expensive to filter through all of the design documents for the impact of a change [MCBR02].

In distributed development environments, an evolution control system must support collaboration between multiple users at multiple sites, support concurrent updates that split development threads into parallel variants, provide mechanisms for notification when changes made by one developer affect the work of another, and when appropriate, provide guidance for decoupling or serialization when on-going work of one developer would be counter-productive to attempted work by another. The artifacts to be controlled in the holistic framework vary in both purpose and format. Examples include organizational policy and vision documents, business case documents, development plans, evaluation criteria, release descriptions, deployment plans, status assessments, user's manuals, requirements and specifications, customer interviews, meeting minutes, code, software documentation, software architecture documentation, unit tests, test cases, and test results. The formats of these artifacts include data base entries, text documents, spreadsheets, images, drawings, audio files, and video clips. A long-term goal of the HFSE is to establish positive control and integration over this diverse set of information.

By relating inputs and outputs of various software process models through an evolution interface that attaches and records the dependencies among evolution artifacts, information required by various processes can be automatically generated and obtained as needed. Such a model requires interaction between a GUI, an evolution control component, and an object model component. The holistic framework can be viewed as

an abstract layer of activity that interacts with subordinate software development tools via middleware communications mechanisms (see Figure 3).

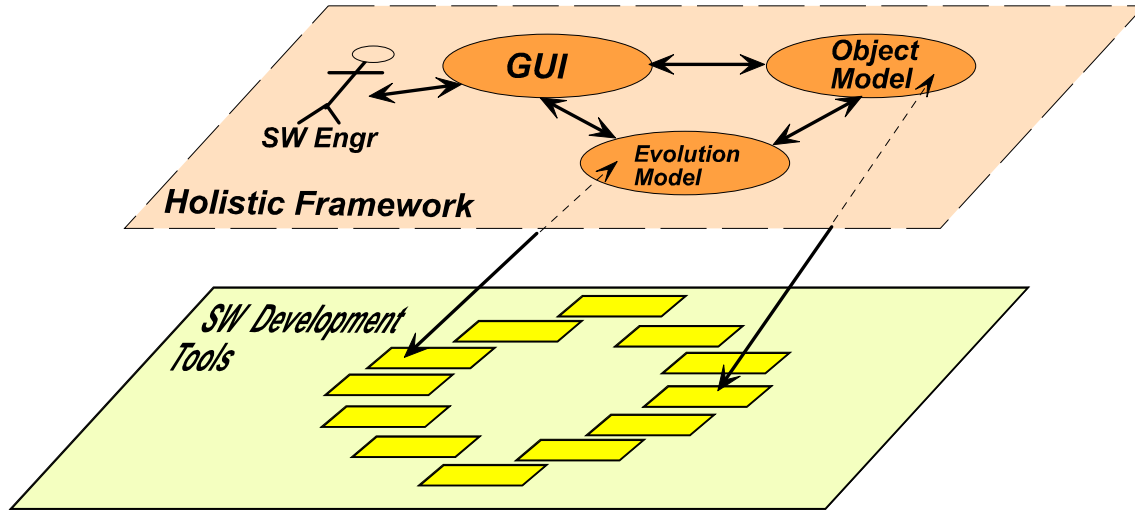


Figure 3 Holistic Model of Software Process Interaction

In the HFSE, the software engineer does not serve as a central bridge between subordinate software development tools, in contrast to Figure 2. However, [MCBR02] pointed out earlier that totally removing the engineer from the development process and automating all central processes is impossible. Thus, the HFSE as presented in this dissertation will only partially achieve the configuration shown in Figure 3.

The Evolution Model and Object Model interact with subordinate software development tools and processes. There are numerous research considerations that must be addressed when establishing this higher-level holistic framework. These include identifying standards for representation and interpretation of information, establishing the medium of communications, accounting for process order, providing missing data, accounting for ambiguity of inputs and outputs, accounting for conflict resolution between models, and providing for extensibility.

The evolution interface was developed so that it can automatically deploy a range of artifact dependencies throughout the lifetime of a particular software project. The interface was established by extending the preexisting Software Evolution model [HARN99c] with Quality Function Deployment (QFD) to introduce a continuum of

dependencies between software artifacts [HAUS88] [HAAG96]. The preexisting model relies on predefined artifacts and limited dependency tracking. A QFD continuum separates relevant dependencies/priorities from noisy data and is an improvement over the previous model [HARN99c] that only provided primary and secondary dependencies with no articulation as to importance or strength of the dependency to the rest of the design. The HFSE also distinguishes the types of the dependencies and provides semantics and standard interpretations for the dependencies to enable tools to take automatic actions based on them. These extensions improve the vertical, horizontal, and temporal dependency graph between software artifacts (e.g. horizontal: requirement 1.2 to requirement 1.2.1; vertical: specification 1.2 to code segment 3.4; temporal: reuse component 4.2 of version 1.0 to reuse component 4.2 of version 1.1).

2. Object Model

The interaction framework between the subordinate process models and the extended evolution model was constructed using the Object-Oriented Model for Interoperability (OOMI) for resolving representational differences between heterogeneous systems [YOUN02b]. This approach establishes a high-level Federation Interoperability Object Model (FIOM) that facilitates the interaction between the objects of existing heterogeneous systems. By establishing such an object federation between existing process models (or their tools) and then integrating that federation with the extended evolution model, inputs and outputs between the subordinate models (or tools) are available to each other while at the same time reporting interaction to the extended evolution model. The framework ontology provides mappings between the subordinate models that preserves those properties that the associated tools depend on. This research helps clarify some of the tradeoffs between interoperability via conformance to a single global data standard versus the use of multiple representations, ontologies, and translations as supported by the FIOM approach. This approach appears viable because global agreements on standards are nearly impossible to achieve in complex domains like software development, and they appear to be unnecessary. The FIOM ontology is used to identify the localized agreements and correspondences necessary at each tool boundary. This approach works because the localized agreements are independent of each other and each is much simpler than the effort required to establish a global data standard. In

addition, this approach accommodates local representations that are optimized for particular tools.

Once the evolution model was extended and an interaction framework established, it was possible to improve the efficiency and effectiveness of software development in a number of ways. First, the entire process of software development became more automatic. As long as model/tool inputs and outputs are supplied through the holistic model, different tools were able to interact automatically, with less involvement by the software engineer. Second, because all artifacts within the holistic model are tracked together as a large dependency graph, it was possible to extract select "slices" of the dependency graph for particular purposes, allowing "focused" software development. For example, since the holistic model interacts with existing process models for software risk management, it was possible to extract a "slice" of the entire dependency graph (a slice that represents the greatest risk) so that prototyping and analysis effort was not wasted on developing artifacts that were already well defined, understood, and/or were successfully implemented in previous versions.

3. The Ideal HFSE

This dissertation research represents an initial investigation into the characteristics needed to establish the HFSE. As such, it is useful to consider all the ideal characteristics of a Holistic Framework and then to discuss that subset of these characteristics that will actually be addressed (in whole or part) in this initial investigation.

a. Generic

The HFSE should be generic and non-proprietary. The framework should allow any model or tool to be incorporated. The framework should not be established solely for use with a specific group of tools.

b. Real Tools

The HFSE should support real software development tools. The framework should not be established simply to support research/laboratory software development tools but must account for tools used to build real software.

c. Process Independent

The HFSE should be independent of the software development process. The framework should be of benefit irrespective of the software process models employed..

d. Domain Independent

The HFSE should be applicable to linking together tools irrespective of the software domain to which they are applied.

e. Extensible

The HFSE should be extensible. Not only should it be possible to include new process models or tools by using the framework, but it should also be possible to modify or update the attributes of the framework based on new technologies or new attributes required by new process models. The ideal HFSE will not have attributes that must be modified when it is extended - the goal is to enable extensions that refer to only adding more attributes, without changing the information you have, except possibly dropping some that may not be relevant in some contexts.

In order to maximize profit, software applications are required to effectively operate for years. Such longevity requires consideration of software tools and languages that also have similar longevity -- or if that is not possible, then consideration must be made to provide a stable long-term development environment in which tools are replaced when required. [MCBR02] states:

Long-lived applications require long-lived development tools... A key question that needs to be asked when considering long-lived applications is whether your development tools are likely to remain stable for the lifetime of the application. This question can be really tough to answer, but we need to start thinking about it because the 1990s were littered with tools that have not survived. How would you act differently if your users insisted that the application has to be useable for the next 20 years, just like any other capital asset?

Thus, the HFSE should provide an environment within which tools can be switched out or replaced to provide developers a long-lived, stable support environment for long-lived applications.

f. Improve Time to Market

Use of the HFSE should improve the likelihood that software is produced in a timely manner. It should be possible to demonstrate that application of the HFSE to a set of development tools will actually decrease the amount of time that is required to produce and implement specific software functionality of interest.

g. Decrease Cost of Development

Use of the HFSE should decrease the cost of developed software. It should be possible to demonstrate that application of the HFSE to a set of development tools will actually decrease the cost required to produce and implement the target software functionality.

h. Improved Quality

Use of the HFSE should increase the quality of developed software. It should be possible to demonstrate that application of the HFSE to a set of development tools will actually produce software that is of better quality than software produced by tools that have not been subject to integration by the HFSE.

i. Easy to Use and Enhances Productivity

The HFSE should be intuitive and easy to use. It should be possible to demonstrate that the HFSE is easy to use with attendant high-levels of productivity.

4. Scope of the Dissertation Research

Of these ideal HFSE characteristics, only items “a” through “e” will be directly addressed and confirmed by this dissertation research. Items “f-i” will be discussed in the dissertation, but formal validation of these characteristics is left to future research.

D. CONTRIBUTIONS PROVIDED BY THIS DISSERTATION

1. Accomplishment of the Research Goal

This dissertation provides confirming evidence of the research hypothesis. In particular, this dissertation demonstrates the following:

a. Construction of the HFSE is Feasible

It is feasible to establish a Holistic Framework for Software Engineering that consists of an extended Software Evolution model integrated with a Federation Interoperability Object Model of subordinate software development models and tools.

b. HFSE Artifacts Can be Described Mathematically

It is feasible to mathematically describe the HFSE constructs.

c. The HFSE Increases Software Tool Interoperability

The application of the HFSE to a sample set of software development tools (i.e., Requisite®Pro and SEATools) increases the interoperability (i.e., data exchange and joint task execution) of the selected set.

2. Other Original and Unique Contributions

While the most important original contribution to the field of software engineering that this dissertation provides is to establish the feasibility of the HFSE, there are several other contributions.

a. Develop a Software Development Tool Ontology Construction Methodology

This dissertation provides the blueprint for building a software development tool ontology. The methodology was adapted from other sources (notably [USCH96]), but was tailored for identifying and capturing the unique characteristics of software development tools. This methodology can be used to add software development tools to a tool ontology.

b. Construct a Pilot Software Development Tool Ontology

This dissertation presents a partial software development tool ontology. Three separate ontologies (and their inter-relationships) are presented: a high-level software development tools ontology, an ontology that describes the Common Object Model (COM) interface of Requisite®Pro, and an ontology that describes important (from an interoperability viewpoint) classes from SEATools. Together, these ontologies form the basis for the Federation Entities in the software development tool FIOM.

c. Adapt QFD Methodology to Deploy Software Dependencies other than Quality

This dissertation demonstrates a methodology for deploying definable software dependencies throughout a software development effort. To date, the main software dependency deployed using QFD has been a customer's view of quality. While the theoretical deployment of other dependencies (e.g., cost, reliability, new technology,

security) have been suggested by other authors, there are no published proposals for deploying these other dependencies. This dissertation presents such a proposal.

d. Apply OOMI to the Software Development Tool Domain

The OOMI was applied to an entirely different domain (other than C4I systems) by establishing a FIOM between software development process models and tools. This effort provided an appreciation of the difficulties in applying Young's methodology to a set of legacy heterogeneous software systems.

e. Use the HFSE to Provide Perspective Views of the Development Effort

This dissertation provides tool support that provides user defined perspective views of particular aspects of a software development effort. These views allow the user to glean important decision support information from the underlying hypergraph of the software development effort. Such decision support can be later shown to provide software process and product improvements.

E. ORGANIZATION OF THE DISSERTATION

This dissertation is organized to progressively take the reader from the theory underlying the HFSE, through the research that led to the development of the framework, and through the efforts to provide tool support for the effort. The dissertation culminates in the application of the framework against a non-trivial software development scenario.

Chapter II presents a survey of previous work on developing interoperability among heterogeneous software development tools. The chapter consists of two main sections. First, a "Foundation Work" section identifies the key research upon which the HFSE research is founded. Examples of this foundation work include software evolution, the Relational Hypergraph model of software evolution, Quality Function Deployment, and the Object-Oriented Methodology for Interoperability. The second major section in this chapter covers "Related Work" -- the areas of research that are in some way in competition with the premise of the HFSE. Examples of related work include software development tool suites, unified processes, and other efforts to form Integrated Software Development Environments (ISDEs).

Chapter III provides the methodology and work associated with establishing the software development tool ontology upon which the HFSE depends. It lays out the methodology used to identify and organize the terminology of the software development tool domain. It presents the captured ontologies of the two specific tools used in the HFSE research.

In Chapter IV the integration of the Relational Hypergraph Model of Software Evolution and Quality Function Deployment is discussed. The chapter explains the theory behind both models and lays out the mathematical relationships of their integration and the mathematical basis upon which the user-defined perspective views are formed from the underlying hypergraph of the software development effort.

Chapter V introduces how the OOMI methodology was applied to the domain of software development tools. It presents the construction of a partial FIOM between Requisite®Pro and SEATools. It discusses the difficulties associated with intralingual translation approaches and what is required to extend the software tool FIOM to include other development tools such as Rational Rose.

Chapter VI presents the resulting HFSE. It describes the HFSE and lays out the way in which a software engineer would apply the HFSE in a given software development scenario. The chapter discusses how the HFSE could be extended to incorporate additional software development tools. Finally, the chapter presents the theory behind how the HFSE could be applied in a given situation to provide an engineer with decision support information related to the development effort.

Chapter VII introduces the tool support of the HFSE (CASES). It explains what version 1 of CASES did and provides details on the many enhancements to CASES so that it now supports application of the HFSE. Examples are shown of how to graphically develop project schemas, register components with external software development tools, construct complex QFD dependency matrices and finally, take slices of the underlying Relational Hypergraph to provide a meaningful subgraphs that form the basis of the decision support the HFSE provides.

Chapter VIII presents the Hello World and CARA Infusion pump software scenarios. The chapter lays out how the HFSE was applied in the scenarios to provide confirming evidence of the dissertation hypothesis.

Chapter IX concludes the dissertation by providing a summary of the contributions of the work as well as pointing researchers towards areas of future investigation.

THIS PAGE INTENTIONALLY LEFT BLANK

II. PREVIOUS WORK

A. CHAPTER ORGANIZATION

This chapter is organized into two main sections: "Foundation Work" and "Related Work." The "Foundations Work" section focuses on that previous research that establishes the underpinning of the investigations presented in this dissertation. The purpose of the section is to provide a starting place, background, and basis upon which the contributions of this dissertation are founded. The "Foundation Work" section is organized into the following areas:

- Approaches to Software Evolution,
- The Relational Hypergraph Evolution Model,
- Quality Function Deployment,
- The Object-Oriented Model for Interoperability, and
- The Use of Ontologies in Establishing Interoperability.

The "Related Work" section of this chapter presents research that attempts to accomplish some of the same aims and objectives as those resolved in this dissertation. The presentation in this section compares and contrasts these other works with the work and accomplishments presented in this dissertation. The "Related Work" section is organized into the following areas:

- Software Development Tool Suites,
- Unified Software Development Processes,
- Integrated Software Development Environments,
- Previous Use of Ontologies in Software Development, and
- Previous Use of QFD in Software Development.

Together, these sections provide a comprehensive literature review of the topics presented in the dissertation and clearly show the basis upon which this work is founded as well as describing how it differs from previous efforts.

B. FOUNDATION WORK

1. Approaches to Software Evolution

Over the years "Software Evolution" has become a recognized and well-established subfield of software engineering. [LUQI89] states that: "Software evolution

refers to all activities that change a software system, including responses to requirements changes, improvements to performance or clarity, and repairs for bugs." This implies that software evolution includes more than just software maintenance, requirements' traceability, and configuration control; it is a recognition that a software product is subject to long-term (birth to retirement) modification in order to meet changes in environment and changes in customer needs. It is also a recognition that all aspects related to this continual, progressive, and evolutionary change may have an impact on how much change is possible, that rate at which change is possible, and how successful that change is in the eyes of the customer.

In his seminal work in researching the problems in the evolution of IBM's software products (and the evolution of the OS/360 in particular), Lehman proposed a new paradigm for dealing with the problems of size and complexity in ever-evolving software systems [LEHM69]:

The [software crisis] implies the need to replace existing unstructured technology-oriented programming methodology by an overall total-process-oriented methodology. This is seen as providing a structure to the process. This structure must be designed to guide the programming process and enable it to achieve any desired combination of the performance, reliability and cost for a minimum in human effort and maximum machine support... By creating an appropriate structure for the process itself, complexity and cost are reduced, and human and machine effectiveness simultaneously increased.

It is interesting to note that even as early as 1969, Lehman recognized the need for taking a holistic approach to software development, recognizing that it was not enough to simply improve individual aspects of the software development process or individual software development tools. He stated that the software development process "is itself a 'system' involving many people, many phases, many components and many requirements. A characteristic feature of systems is that local subsystem optimisation does not lead to system optimisation. Thus investigation and improvement of a part of the process without consideration of the process as a whole demands extreme care" [LEHM69]. Consequently, it is through the development of holistic development environments (such as that provided by the HFSE) that significant progress can be made in providing

enduring customer satisfaction in the face of the inevitable evolutionary change that accompanies any software project of significant size and complexity.

Much of Lehman's work has focused on treating large, long-duration software products as feedback and control systems subject to real-world forcing and damping functions (e.g., new customer demands, developer's insertions of new code and modification of existing code, budgetary constraints, changing technology, etc.). To illustrate Lehman's concept, consider an imaginary software system from birth to death, as depicted in Figure 4.

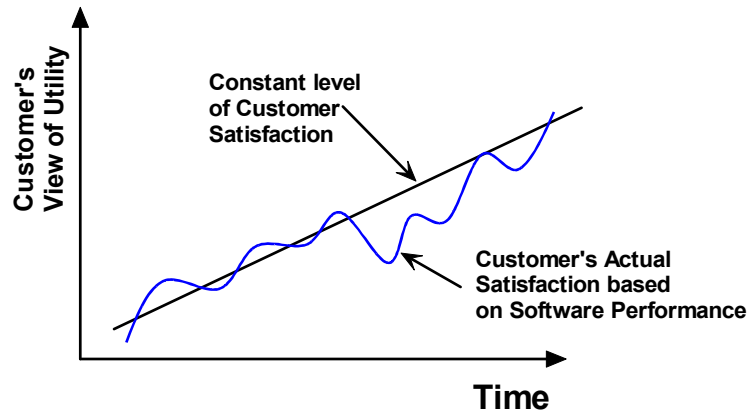


Figure 4 Software Evolution as a Feedback and Control System

From the developer's viewpoint, the developer is always trying to attain and maintain a constant level of customer satisfaction. From the customer's viewpoint, that constant level of satisfaction translates into an ever-increasing amount of demanded utility from the system. While over time, demanded utility is shown here as linearly increasing, this could actually be increasing non-linearly based on market competition and new user needs. The software that is delivered determines the customer's actual satisfaction level (how useful it is in comparison to the customer's needs, how it matches the customer's expectations, etc.). At the same time the actual level of satisfaction is negatively influenced by both internal factors (e.g., defects, poor documentation, etc.) and by external factors (e.g., competitor's products that are perceived to be better, obsolescence of hardware, etc.). Unfortunately while these internal and external forces are not necessarily bounded, it remains incumbent on the developer to account for them anyway: "Exploiting

this multidimensional unboundedness is an inevitable consequence of computer usage" [LEHM98].

Much of this "unboundedness" is caused through imprecision and obsolescence of natural language expressions of stakeholder desires, as well as the problem of stated but misunderstood, improperly stated, or unstated assumptions. [BERZ91] points out that one of the main challenges of software evolution in traditional contexts is the lack of accuracy in requirements, specifications, and design documents. [LUQI89] continues by pointing out that the developer must have precise documentation to reliably change the system. Unfortunately in the case of older systems, such documentation (other than the source code) typically does not exist, has been lost, or is obsolete because of the large amount of time and effort required to manually create and maintain it. In the case of assumptions [LEHM98] observes that:

The sources and nature of the assumptions are countless (I have estimated that a typical program has about one real-world assumption for every 10 lines of code). Some of these assumptions will remain valid throughout the system's life; others will be invalidated by subsequent changes in the application or its operational domain. Still others will fall somewhere in-between: valid in some circumstances, but leading to unacceptable results or behavior in others. Such invalidity generally remains undetected until a problem arises or a disaster...

An additional source of unboundedness arises from what [LEHM91] terms as "*Gödel-like*" and "*Heisenberg-like*" uncertainties. In Gödel-like uncertainty, the real-world software system (including its users and developers) can be viewed as a model of some desired functionality operating in an infinite universe. The users and developers (who are actually part of and internal to the model) cannot completely know the properties of the model and its interaction with the external universe -- their knowledge is incomplete. In Heisenberg-like uncertainty, implementation of the software system changes the operating environment of the system. Once users gain first-hand experience of the system, their expectations of what the system is supposed to do fundamentally change. Unfortunately, total control of assumptions and uncertainty can never be achieved; but, "there can and must be significant improvement in the manner in which such matters are controlled in current industrial practice" [LEHM91].

One way to tackle this problem is to do a better job of recording application and domain boundaries. [LEHM98] suggests that such decisions should be recorded in a structured fashion that also displays the recognized dependencies and relationships between them. Developers must recognize, capture, and record assumptions, whether explicit or implicit, in design and implementation decisions. They must also record any dependencies and the relationships between them. The HFSE provides the developer a framework for recording these assumptions and dependencies. But it is not enough to simply record these assumptions and relationships, proposed changes to a software system must be examined in relation to the assumptions to ensure that intended behavior is achieved.

Lehman's continuing and voluminous body of work in the field of software evolution has led to the development of what are now known as Lehman's Laws of Software Evolution, summarized in Table 1 below [LEHM97].

# (Yr)	Title	Description
I (74)	Continuing Change	Software systems must be continually adapted else they become progressively less satisfactory.
II (74)	Increasing Complexity	As a software system evolves its complexity increases unless work is done to maintain or reduce it.
III (74)	Self Regulation	A software systems' evolution process is self-regulating with distribution of product and process measures close to normal.
IV (80)	Conservation of Organizational Stability (invariant work rate)	The average effective global activity rate in an evolving software system is invariant over the product lifetime.
V (80)	Conservation of Familiarity	As a software system evolves all stakeholders (e.g., developers, sales personnel, users, etc.) must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
VI (80)	Continuing Growth	The functional content of software systems must be continually increased to maintain user satisfaction over their lifetime.
VII (96)	Declining Quality	The quality of software systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

# (Yr)	Title	Description
VIII (’96)	Feedback System	Software evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

Table 1 Lehman's Laws of Software Evolution (after [LEHM97])

These "laws" and years of observation have led Lehman and others involved in the FEAST (Feedback, Evolution, and Software Technology) Research Group to begin work in trying to define a theory of software evolution, consistent with empirical evidence collected to date [LEHM00]. FEAST has been primarily concerned with the "*properties* of the evolution phenomenon, the *what* and the *why* of evolution." The group has sought to understand the software evolution phenomenon by applying the scientific method of observation, measuring, modeling, interpretation, and hypothesis generation with a primary goal to "determine the underlying *causes*, *attributes* and practical *impact* of evolution" on software development processes and products. This is different than the view taken in this dissertation in which the concern focuses on "the *how* of evolution" and in particular on how the improvement and management of the evolution process can lead to higher productivity, improved quality, faster development, greater adaptability and reliability, etc. "Those interested in the *why* and the *what* see evolution as the sum total of activity required to maintain stakeholder satisfaction over application lifetime. Those that focus on the *how* see it as the *process* to achieve satisfactory, controlled and disciplined software change" [LEHM00].

In summary, there are five important concepts in how others have tackled the domain of software evolution that are important to this dissertation. Each of these is listed below with the implication to the HFSE highlighted.

a. *Inevitability of Evolution*

Software of any significant size or complexity will inevitably require modification in order to continue to provide customer utility. **HFSE implication:** The HFSE tracks and records many artifacts and potentially even more relationships between

artifacts over great periods of time. The framework provides the software engineer an easy and efficient way to "browse" through these artifacts and relationships.

b. Holistic Approach

Improvements to individual parts of the software development process without regard to the whole process do not guarantee overall improvement in process or product. **HFSE implication:** The HFSE is holistic in nature. It is able to track artifacts and relationships throughout the entirety of the software development effort. It provides the engineer the ability to view specialized information across this entirety.

c. Assumptions and Uncertainty

Imprecise and inadequate documentation, uncertainty and incompleteness of software modeling, and unstated/misstated/obsolete assumptions create a natural decay in software quality. Lehman's 2nd, 6th, and 7th Laws imply that preventing the decay requires more and more of the total software maintenance effort over time. **HFSE implication:** The HFSE provides the engineer the ability to recognize, capture, and record assumptions, whether explicit or implicit, in design and implementation decisions. Furthermore the HFSE provides the engineer the ability to record assumption dependencies and the relationships between the assumptions and other software artifacts. It is not enough to simply record these assumptions and relationships, the HFSE allows the engineer to examine proposed changes to a software system in relation to the assumptions to ensure that intended behavior is achieved. Unfortunately, the HFSE does not directly correct the "unstated or mis-stated" assumption problem; however, it does provide the engineer a powerful framework for isolating and then identifying such problematic assumptions.

d. Validation Against Assumptions

Software engineers must have better mechanisms that allow them to continually validate the current state of the software against previously stated (and unstated/mis-stated) application boundaries and assumptions. **HFSE implication:** the HFSE provides the engineer the ability to select specific subsets of information traceable to the current design. These "views" give the engineer insight into how their current software artifacts match up against previously produced artifacts (including assumptions).

e. *Views of Evolution*

Software Evolution can be seen from two points of view: those interested in *why* evolution occurs and *what* causes it, and those interested in *how* evolution can be managed to produce improvements to software processes and products. **HFSE implication:** The HFSE seeks to implement the second view of evolution. It provides a framework by which software processes and products can be improved (however, recall that this dissertation only seeks to prove the feasibility of the HFSE and proof of improvement is considered future research).

The implication of these five concepts to the development of the HFSE is significant. Software evolution forms the core of the framework providing the software engineer a mechanism by which he can record assumptions, relationships, and dependencies and then easily isolate important decision support information.

2. The Relational Hypergraph and CASES

a. *Brief History of the Hypergraph*

Claude Berge first introduced the hypergraph in 1960 as a way to generalize graphs [BERG89]. In particular, he defined a hypergraph \mathbf{H} as shown in Equation 1.

$$\mathbf{H} = (\mathbf{V}, \mathbf{E})$$

where

$$\mathbf{V} = \{v_1, v_2, \dots, v_n\} \text{ is the set of } n \text{ vertices (or nodes)} \quad \text{Equation 1}$$

$$\mathbf{E} = \{E_1, E_2, \dots, E_m \mid E_i \subseteq \mathbf{V} \text{ for } i = 1, \dots, m\}$$

is the set of m hyperedges

An undirected hyperedge E_i represents a relationship involving all the vertices in the subset of E_i . Given this definition, it is easy to see how this generalizes previous definitions of graphs, because a typical graph where each undirected edge connects two nodes would be the special case of a hypergraph such that the condition in Equation 2 holds.

$$|E_i| = 2 \quad \forall i \quad \text{Equation 2}$$

In a *directed* hypergraph, two additional functions are defined on the hyperedges ($T(E)$ and $H(E)$) to distinguish the "tail" vertices of the hyperedge and the "head" vertices. It is

then straightforward to produce an incidence matrix $\mathbf{A} = (a_{ij})$ defined by Equation 3 that records all necessary information about the hypergraph.

$$a_{ij} = \begin{cases} -1 & \text{if } v_i \in T(E_j) \\ 1 & \text{if } v_i \in H(E_j) \\ 0 & \text{otherwise} \end{cases} \quad \text{Equation 3}$$

Figure 5 illustrates a directed hypergraph with its associated incidence matrix.

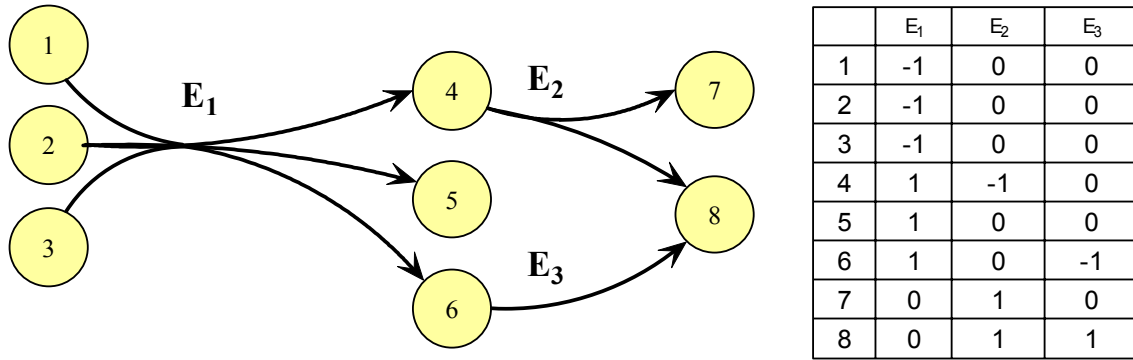


Figure 5 Directed Hypergraph with Incidence Matrix

Since the introduction of hypergraphs, hypergraphs have been used in numerous cases in combinatorics and computer science and more recently in the area of software evolution.

b. The Relational Hypergraph (RH) Software Evolution Model

Harn, in his PhD dissertation [HARN99c], extends the work of several others [BORI86], [LUQI90], [BADR93], and [IBRA96] in establishing a *Relational Hypergraph model (RH model)* to describe software evolution. This model establishes dependencies and links between key activities/artifacts of a software development cycle and also between sequential iterations of cycles. The model allows the development of tools to manage both the activities in a software development project and the products that those activities produce. An example of such a tool is the Computer Aided Software Evolution System (CASES) developed at the Naval Postgraduate School in support of Harn's work.

CASES, programmed in Java, is a software system that performs the following key functions during software evolution: control, management, formation,

refinement, traceability, and assignment. CASES manages and controls all of the activities that change a software system and the relationships among these activities. CASES is based upon the relationships in the evolutionary process model shown in Figure 6.

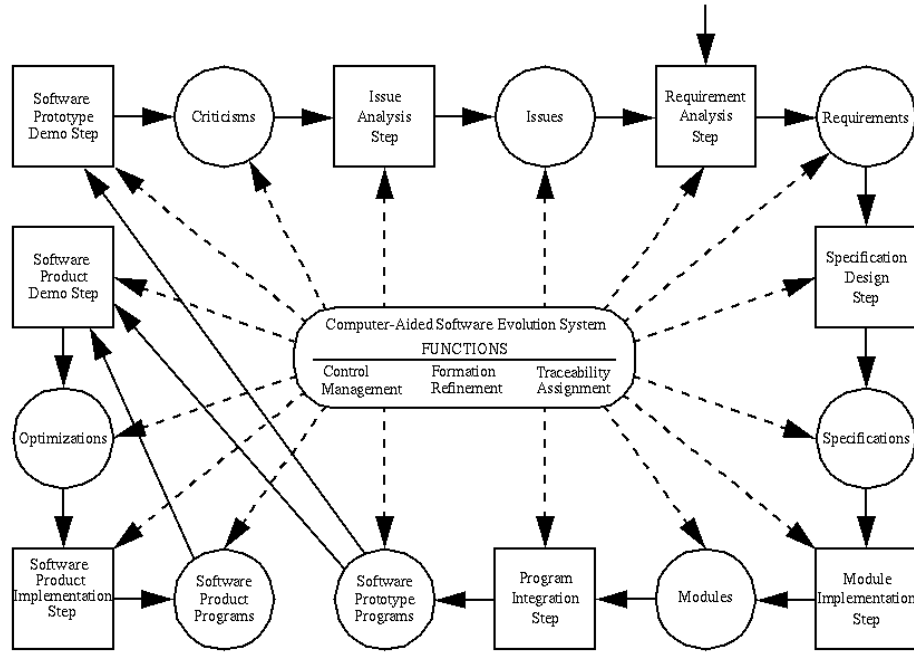


Figure 6 Software Evolution Processes with CASES (from [HARN99c])

Underlying CASES are software evolution models that use the relational hypergraph to mathematically characterize the relationships between software activities and artifacts. Harn adapts Berge's hypergraph [BERG89] to establish a hypergraph model, evolutionary hypergraphs, and relational hypergraphs as follows [HARN99c]:

A hypergraph model represents the evolution history and future plans for software development as a hypergraph. Hypergraphs generalize the usual notion of a directed graph by allowing hyperedges, which may have multiple output nodes and multiple input nodes.

An evolutionary hypergraph is a directed, labeled hypergraph that has been annotated with the attributes of the evolutionary components and steps of a software development process.

A *relational hypergraph* is an evolutionary hypergraph in which the dependency relationships between components and steps can have a hierarchy of specialized interpretations.

In the Relational Hypergraph, activities and artifacts affected by the software evolution process are called software evolution objects and consist of "Steps" and "Components." Harn identifies eight types of steps: the software prototype demo step (s-C), issue analysis (s-I), requirement analysis (s-R), specification design (s-S), module implementation (s-M), program integration (s-P), Software product demo (s-O), Software product implementation (s-Pd). There are eight types of components: Criticisms (C), Issues (I), Requirements (R), Specifications (S), Modules (M), Software prototype programs (P), Optimizations (O), Software product programs (Pd). The Relational Hypergraph model uses a hierarchical refinement (top-level objects, refined objects, atomic objects) to link these objects and establish dependencies (both primary dependencies and secondary dependencies) between the objects.

Harn's work forms the basis for establishing a Software Evolution Model as the core for the HFSE. By reworking the evolution model to become more extensible, the Relational Hypergraph becomes a very useful mathematical construct for establishing dependencies between evolution artifacts.

Harn's work has two primary weaknesses when viewed from a perspective of its usefulness to developing a Holistic Framework for Software Engineering.

(1) Lack of Generality & Extensibility. Harn's work was established for a single process model -- the evolutionary prototyping model. While he demonstrated the use of this model on different domains of systems (C4I and embedded real-time systems) he did not demonstrate its applicability for use with other main-stream process models (e.g. spiral development, waterfall, Win-Win, etc.). Also, while the original graph model is extensible [LUQI90] (Harn's model being one of those extensions), Harn failed to provide a uniform method for extending the base model to support new process models. While it is fairly easy to add and modify components, steps, and attributes in his model, many of the rules governing their relationships have to be modified as well. The lack of generality is perhaps the reason why the model has not been widely adopted in industry.

Overcoming the lack of generality in Harn's evolution model requires that the model be redone within a generic framework. Constructs are included in the HFSE that allow the software designer to "build" the objects, components, steps, and attributes that the designer actually uses in their specific software development process and then to link these objects together through establishment of dependency relationships using QFD.

(2) Parametrical dependencies. Harn established the Relational Hypergraph model to only account for a limited number of dependencies between artifacts (e.g. Primary-input driven and Secondary-input driven). Unfortunately, this does not match the real world in which software artifacts are related through a continuum of dependencies. The HFSE defines and implements a small finite set of distinct meanings for dependencies.

Embedding QFD into the model solves this shortcoming. QFD allows for and keeps track of a continuum of dependencies between objects (in fact, it is the continuum of dependencies that form the core of the QFD methodology).

3. Quality Function Deployment (QFD)

a. QFD History and Cited Benefits

QFD was originally developed in Japan in the mid 1960's as a "requirements based" methodology that ensures that the "Voice of the Customer" is deployed throughout the product design and manufacturing process. The Japanese application of QFD in the 70's is often cited as a primary factor that allowed them to dominate the global automobile industry for almost a decade and in the early 80's it was the US application of QFD that allowed the US automobile industry to recapture some of its lost global market-share [CLAU88] [HAUS88]. Since then, the use of QFD has been extended to the entire US manufacturing industry as a means of ensuring products meet customer requirements.

QFD establishes a conceptual map that provides the means for inter-departmental functional planning and communications [HAUS88]. The customer's own words and phrases are captured and used whenever possible. It is these words that are simultaneously translated by the designers, architects, programmers. One of the many

benefits of QFD is to enhance cross communications between departments in the same development effort [COHE95]; consider Figure 7.

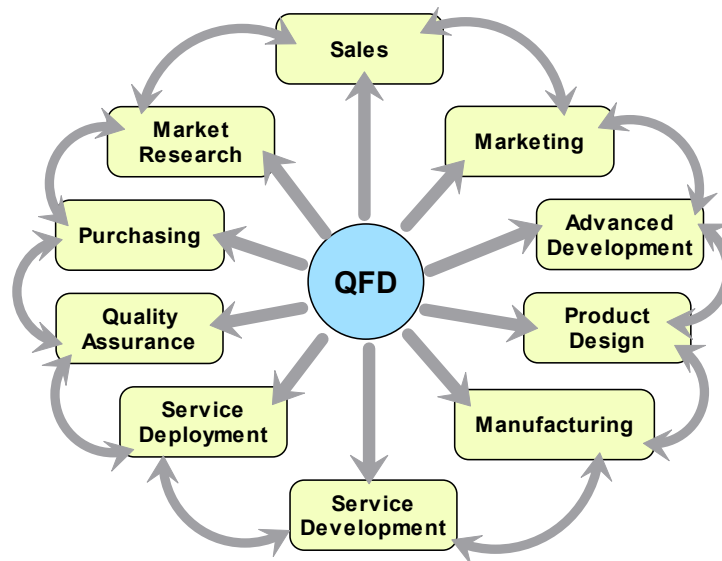


Figure 7 QFD Enhances Cross-Functional Communication (after [COHE95])

Recall the likeness of this diagram to the traditional software development diagram presented earlier (Figure 2). Because QFD acts a central communications mechanism for deploying the customer's vision of a quality product across different functional areas in a company, in many ways, QFD can be used to perform much of the dependency "bridging" that has traditionally rested on the shoulders of the software engineer.

QFD can also help to improve product development processes. All too often, specialists in particular areas focus on improving their particular area of the process while losing sight of the holistic nature of the development effort. [CLAU88] states:

When specialists polish their own specialties within their own segments, their activities may appear very elegant and impressive. But all too often, they simply lead to "institutionalization of waste."

An example of this was the highly touted development of automated storage and retrieval systems (AS/RS) that were to revolutionize the warehousing industry. While such systems significantly improved warehousing, they became virtually useless with the advent of "Just-in-Time" inventory processes in which the overall goal was not to speedup warehousing activity, but to eliminate it altogether. What is needed in such

cases is greater cross communication between development segments -- a need that QFD is designed to address.

b. Software Quality Function Deployment (SQFD)

While QFD has been widely and successfully integrated into much of the U.S. product industry, the same cannot be said of the integration of QFD into the U.S. software industry. There are a number of reasons for this lack of integration. A Japanese pioneer of QFD, [AKAO90] comments that "the quality function deployment method for software development is not yet well established" and that "examples of the use of quality charts [QFD matrices] for determining the quality plan and quality design are also scarce. Identifying critical functions deployed for individual software is a challenge to be addressed in the future." He states that the much of this problem stems from a general lack of quantitative measurement methods and ways to test and measure software quality. Measuring the frequently cited classifications of software quality characteristics (including objectivity, operability, performance, reliability, usability, confidentiality, security, expandability, interchangeability, reusability, and continuity) has proven very difficult and is a key reason that there has not been more effort to integrate QFD into software development processes.

In discussing the use of QFD for embedded systems, [THAC90] attributes the lack of integration of QFD into software processes partly because of incompatibility between the terminology used in QFD and the terminology used in software. Also, that while many software companies are trying to implement simultaneous (or concurrent) software development processes, they are failing because of a lack of a definable concurrent engineering process. The reason seems to be the lack of a consistent and properly documented process that covers all aspects of software development while simultaneously addressing the discipline of embedded systems development and is tied to the organizational structure of the companies constructing the embedded software. While Data Flow Diagrams and Control Flow Diagrams are used widely in embedded systems design, they have a number of shortcomings [THAC90]:

- They do not show which components map to which customer requirements.
- They do not show potential conflicts in requirements.

- They do not show the positive or negative impact of one design element on a particular requirement.
- They do not show the positive or negative impact of one design element on another design element.
- They do not show which method, tools, and procedures are used for various parts of the design.
- They do not show how the resource plan maps to the development effort.

Existing requirements analysis and specification methodologies (e.g. structured analysis and design, object modeling techniques) do not clearly identify how software development tasks implement what customers want [LIUX00]. Also, existing methodologies fail to identify and resolve conflicts between customer requirements. QFD is a methodology that overcomes each of these challenges when used prior to the creation of Data and Control Flow diagrams. [THAC90] goes further and points out that for QFD to be properly integrated into existing software design processes, a lot more automation of the process is needed. In particular he points out the following:

- Product development information should be freely and seamlessly available across all implements.
- Software Engineers need to be able to 'browse' through engineering data at will and create new relationships where appropriate.
- Other designer should be able to trace back and determine engineering trade-offs, reasons for design decisions, etc.

It is these very issues related to automation that embedding QFD into the Relational Hypergraph model of Software Evolution (forming the HFSE) is designed to solve.

While QFD has not yet been *widely* integrated into software development, there are cases when QFD has been used successfully in software development. [LIUX00] states:

It [SQFD] has been applied to improve software quality in many large organizations, such as DEC, AT&T, Hewlett-Packard, IBM and Texas Instruments... SQFD has been utilized in developing various types of software products, such as operating systems, embedded software, management information systems, decision support systems, network and transaction processing systems. SQFD has been beneficial in developing new software products and upgrading or enhancing existing software products. It helps to enhance communication between customers and software developers and testers.

In the cases where QFD has been used for software development, cited benefits of SQFD include the following [HAAG96]:

- Fosters better attention to customers' perspective.
- Creates better communication among departments.
- Provides decision justification.
- Quantifies qualitative customer requirements.
- Represents data to facilitate the use of metrics.
- Facilitates cross-checking.
- Avoids the loss of information.
- Reaches consensus of features quicker.
- Reduces product definition interval.
- Can be adapted to various Software Development Life Cycle methodologies.

The benefits of SQFD appear to be synergistic and lead to fewer changes in requirements specification, design, and code, and reduction in the number of defects and less rework, and therefore, higher productivity [LIUX00]. As QFD was initially invented in Japan, SQFD was also first invented and used in Japan -- initially to improve the quality of embedded software. As reported in [ZULT93], one of the companies that has incorporated SQFD into their software development processes is NEC's IC Microcomputer Systems Company (NEC IC Micon) the first software organization to win the Deming prize. During the period 1982 to 1987 the 1000 person software organization was able to reduce shipped software defects from 45 to 0.5 defects per million lines of executable code, increase their productivity by 5 times, increase sales by 5 times, thus increasing their market share by 20%, which in turn increased profit by 4 times.

QFD, when applied to software, allows managers to focus precious project resources on the customer's high-value software elements and in the end, produce a better software development process and a better (higher quality in the customer's eyes) software product [ZULT92]. [LIUX00] states: "Software Quality Function Deployment (SQFD) focuses on improving the quality of both the software development process and the product." QFD can be used to deploy not only quality, but also technology, cost, reliability, or any special concerns such as usability, reuse or security. The QFD methodology provides both forward and backward traceability in the development life

cycle. Because the QFD process is flexible, engineers can define other appropriate deployments to address specific concerns of customers or the development organization [ZULT93]. Yet, while [ZULT93] theorizes about the deployment of these other customer concerns (dependencies other than quality), there is no evidence that any formal work in establishing a methodology to deploy such dependencies has yet taken place. One of the goals of the HFSE is do just that.

c. The Voice of the Customer

QFD, in its purest form, is designed to deploy the "voice of the customer" throughout a particular design. The goal is to use the customer's own words to "deploy" the customers' notions of what is of value in the product to every aspect of the development effort of the product. However, there may be some ambiguity (e.g. what did the customer mean by "easy to use?") and while it may take some time to ferret out exactly what the customer meant by such a phrase, the time spent in identifying and implementing the customer's intent will directly leads to a quality product. Key is for designers and engineers to avoid interpreting customer phrases without clear understanding of customer intent [HAUS88]. A designer or engineer's inference could lead the development team to tackling problems that the customer deemed to be unimportant.

The use of QFD in capturing and deploying quality throughout a software product's lifecycle implies a shift in the traditional paradigm of what it means to have a quality software product. [LIUX00] states that: "Software quality can be viewed as conformance to software requirements from customers." Unstated in this quotation is the fact that the customer might not have articulated many requirements and that these requirements remain hidden until after an unsatisfactory product is delivered to the customer. Traditional approaches to software quality have relied on Statistical Process Control (SPC). The aim has been to minimize customer dissatisfaction by removing software defects through appraisals, logging/correcting customer complaints, completing software reviews/inspections/walkthroughs, and performing software testing. SPC aims to minimize the amount of "negative" quality in software by removing such defects. With the addition of error cause removal, it was hoped that through process improvement, the generation of defects (and the likelihood that they would remain

undiscovered until after the software was delivered) would decrease. This approach also seeks to minimize the "negative" aspects of quality; however, such approaches do not guarantee that any "positive" quality attributes still remain in the software. They only ensure that the system is *less* bad, not necessarily good; consider Figure 8.

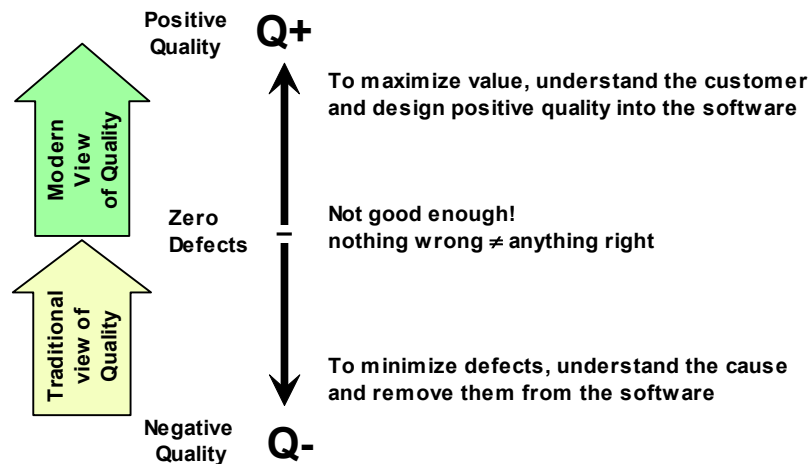


Figure 8 The Quality Continuum (after [ZULT92, 93])

It is a necessary condition that defects are removed from the software, but it is not a sufficient condition [ZULT92]. In addition to SPC, there must be mechanisms built into the software development process that ensure that there are systematic, controlled, and traceable means of ensuring that what the customer considers to be a quality attribute is delivered in the final product. There must be an understanding of exactly what is of value to the customer and then ensure that this value is deployed throughout all aspects of the development effort. [ZULT90] notes that the application of QFD to software will help to both prevent defects, but more importantly, to input positive quality attributes:

Today software engineering using [SQFD] concentrates more on maximizing user satisfaction from the software engineering process. The focus is on preventing the causes of defects by a deeper understanding of the user's true requirements -- starting with a careful study of user and stakeholder wants, needs, and concerns. With [SQFD], you work downstream to design quality into the system, and continuously work to improve the software engineering process with innovation. This approach seeks to maximize the users' complements (positive quality). Only strong positives can make software so good that user boast about it -- the true test of exciting quality.

It is very typical for users to have difficulty in articulating all of their requirements. It is the responsibility of the software engineer to "ask why five times" to define and analyze at a fundamental level the users' requirements [ZULT90]. The engineer must ask why the customer does what he does, why he has the problems he has, and why he is able to take advantage of particular opportunities. The engineer can only develop a complete and consistent set of requirements after getting close to the customer and understanding their wants, needs, and concerns. The engineer must get a sense of how meeting particular requirements effects customer satisfaction. As reported in [AKAO90] [COHE95] and [ZULT90, 92, 93] Kano et. al. provides a model (the Kano Model) that characterizes requirements based on customer satisfaction (see Figure 9).

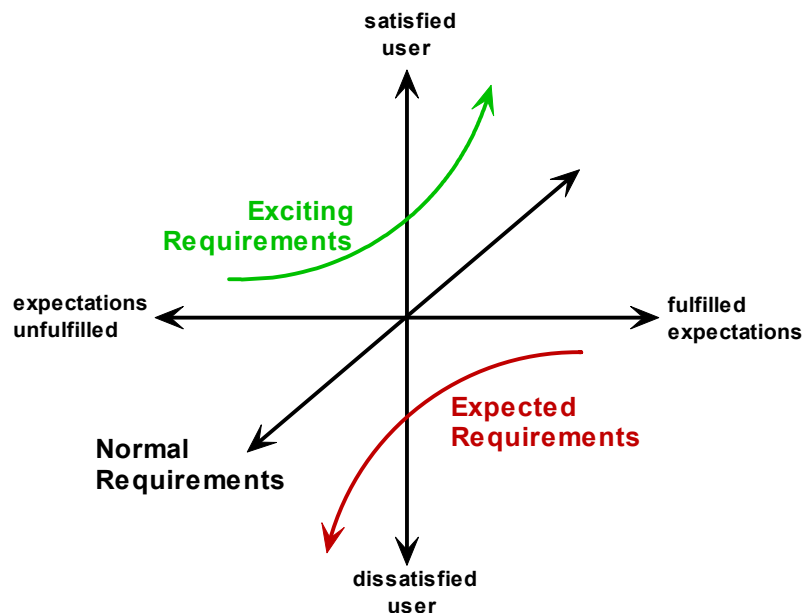


Figure 9 Kano diagram for Requirements (after [ZULT90])

Expected Requirements are the "must" features that a user expects. The presence of these features meets their expectations, but their presence does not necessarily satisfy them. Often, expected requirements are so basic that customers fail to mention them until the software fails to perform an expected function -- the customer is dissatisfied with the result. Normal requirements (sometimes called "revealed" requirements [LIUX00]) vary in proportion to the presence of the feature. The more of a normal requirement is present, the more the customer likes it. An example might be

communications speed -- the faster a system is able to communicate, the more a customer likes it. Exciting requirements are features beyond the users' expectations and are often hidden. The absence of these features generally does not cause any dissatisfaction, because the user was not expecting the feature in the first place. Exciting requirements represent significant opportunity for the developer because if the developer can identify and implement these features, he will significantly add to the customer's satisfaction and may fill an unfulfilled niche in the market. An example of an exciting requirement might be the addition of a new feature enabled by a comparatively new technology of which the user was unaware but the developer had used previously.

[COHE95] asserts that there are two major lessons that Kano's model teaches us. The first lesson is that all customer satisfaction attributes are not equal. Not only are some attributes more important to the customer than others; but, some attributes are important to the customer in different ways than others. For example, requirements that are "dissatisfiers" do not matter at all to the customer when they are met, but they seriously detract from overall customer satisfaction when they are not met. The second lesson is that as [ZULT92, 93] pointed out earlier, the old product quality strategy of responding to customer complaints is inadequate; such complaints are likely to be caused by the failure of a dissatisfiers requirement being met. A quality strategy based solely on removing dissatisfiers can never result in satisfied customers.

The last note about obtaining the "voice of the customer" is that it is generally not sufficient to read reports and conduct customer surveys. The engineer must observe the customer in their workplace to get a true sense of the conditions and requirements of the customer. In the Japanese application of QFD, this is termed as "going to the *gemba*." As stated by [ZULT93]:

We must not be content with an abstract knowledge of the customers' requirements, but acquire a gut-level understanding of the contexts of the customers. Such knowledge does not come from studying a thick requirements document, running focus groups, or facility conference room meetings with customers. It comes only from going to the *gemba*.

Contextual Inquiry (CI) is another technique for gathering customer requirements at the customers' workplace. [HRON93] reports that: "Digital [Corporation] has fostered a

technique called Contextual Inquiry, in which the product developers visit the customer's workplace and observe and interview various users while they are engaged in their normal work activities." Also, [LAMI95] draws the parallels between "going to the gemba" and "CI": "In CI, a fundamental principle is that users are to be studied in their normal working context ('going to the *gemba*.') [sic] Users are studied as they perform ordinary work tasks. Analysts following the CI technique observe users working and record both what the users do and how they interact with their work environment." Whether it is Contextual Inquiry or "going to the gemba", a key portion of the QFD process involves the engineer obtaining a fundamental understanding of all of the customers' requirements by witnessing first-hand the customer in their work environment.

d. Steps in the QFD Process

QFD ensures that the "Voice of the Customer" is deployed beyond the requirements capture phase of the design and is fully embodied in the product specification, architecture, and production phases of the process. QFD is a stepwise process with results recorded in matrices that are sometimes known as "Houses of Quality" because of the characteristic "house" shaped matrix for recording QFD relationships (illustrated at Figure 10).

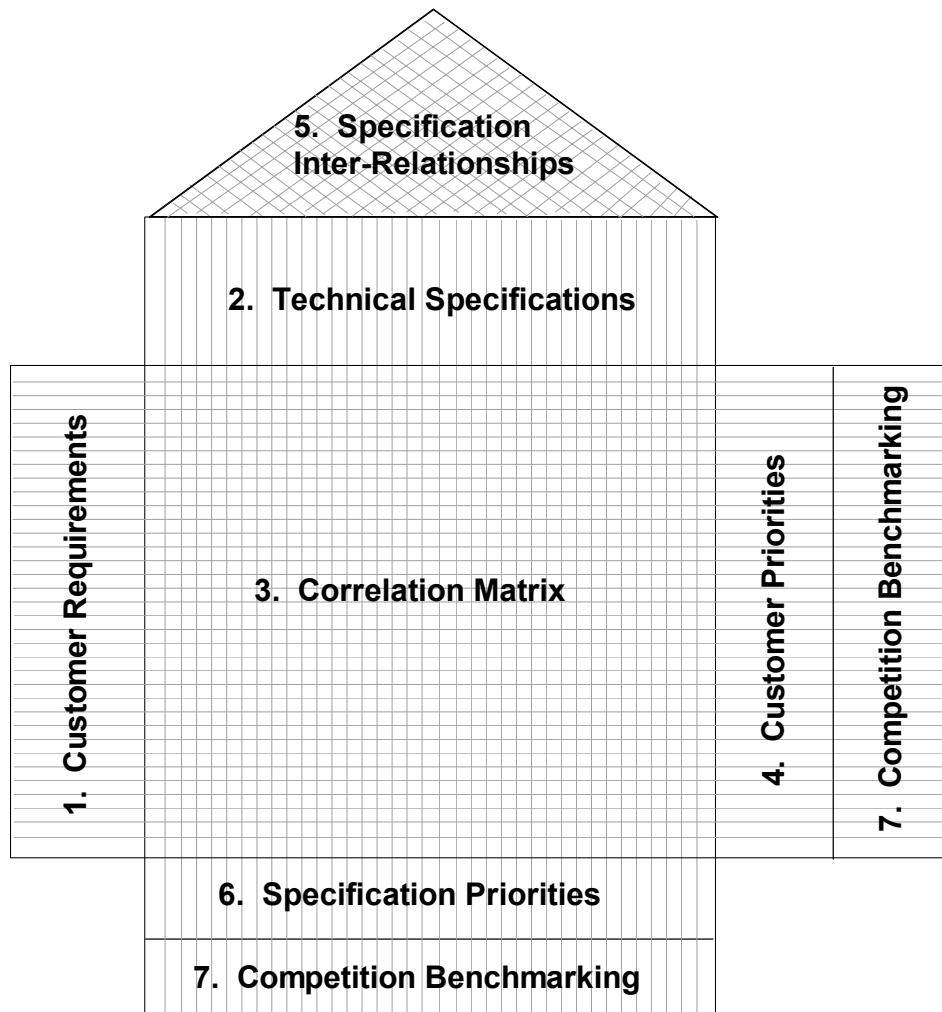


Figure 10 First Level QFD Matrix -- The "House of Quality"

When completing the first QFD "house", QFD consists of the following steps:

- (1) Identify Requirements. Stakeholder Requirements are solicited and recorded on the left y-axis. In more general terms this left hand side is also known as the "Whats" of the design -- where the "Whats" are those items that are desired.
- (2) Identify Technical Specifications. In cooperation with stakeholders, the requirements are then converted to technical and measurable statements of the software product and recorded on the top x-axis as specifications. In more general terms this top row is also known as the "Hows" of the design -- where the "Hows" are the ways in which the "Whats" will be implemented.
- (3) Correlate "Whats" and "Hows." The stakeholders then complete the correlation portion of the matrix by identifying the strength of the

relationships between the requirements and specifications. These values are known as "correlations."

(4) Establish Priorities. Based on the stakeholder surveys (or other analytic means), the priorities for the requirements are established and listed down the right y-axis. In this dissertation these values will be more generally called "dependency values."

(5) Establish Interrelationships. The relationship (and the strength of relationship) between specifications is identified and recorded across the top (the roof) of the matrix. These relationships represent potential engineering tradeoffs between specifications. The engineer balances these tradeoffs in order to optimize the design solution.

(6) Calculate Priorities of the "Hows." Specification priorities are obtained by multiplying the stakeholder requirement priority and the correlation value of specific specifications. These are recorded along the bottom of the matrix.

(7) Competition Benchmarking. Competitors' products can be benchmarked against either the customers' requirements or the technical specifications.

While this first QFD matrix is easily constructed, the real strength of the QFD methodology occurs after the completion of the highest-level matrix. As the project continues, additional matrices are established, each of which establishes dependencies with the original stakeholder requirement priorities. This provides visibility of what is important and what is not. This quality deployment process is illustrated at Figure 11. In the case of a general set of QFD matrices, the "dependency" that is being deployed is a customer's view of "quality." However, in this dissertation that dependency may be defined as a type other than quality (e.g. risk, security, difficulty of implementation, etc.).

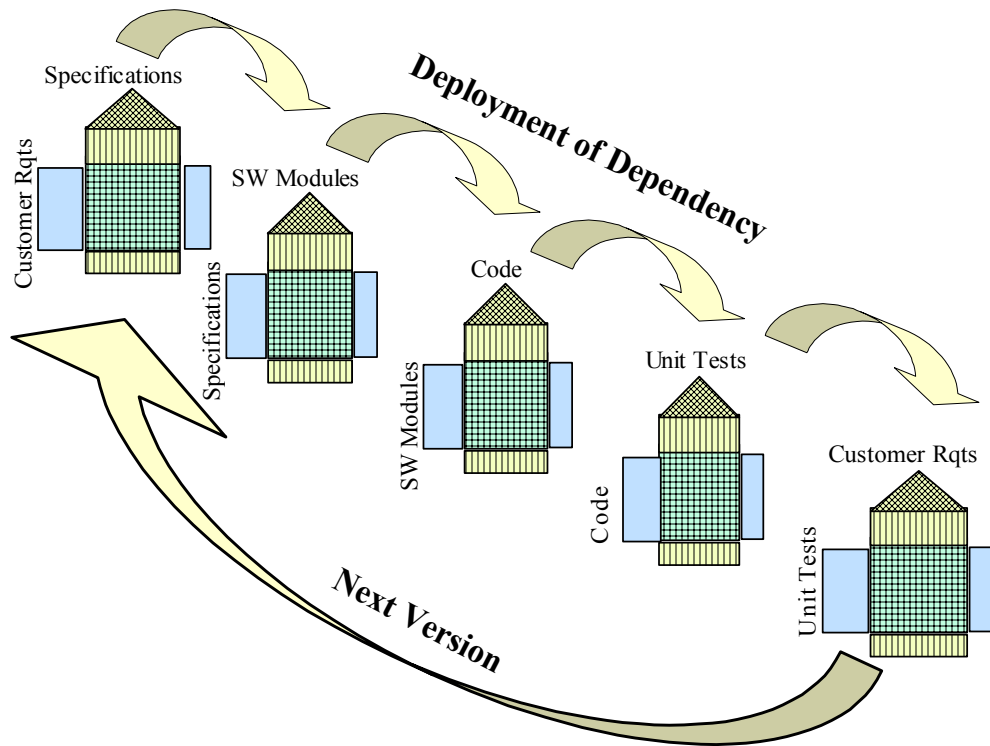


Figure 11 Example of a Simplistic QFD Matrix Deployment

Note that the output of one matrix becomes the input (or starting place) of the next (e.g., customer requirements relates to specifications, specifications relate to software modules, etc.). It is possible to illustrate this set of QFD matrices as a software development process diagram, as shown in Figure 12.

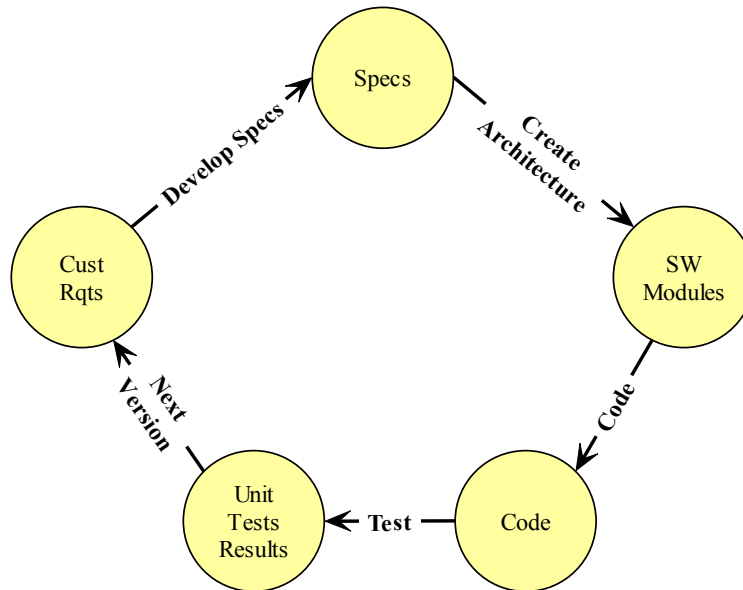


Figure 12 Simplistic SQFD Model drawn as a Process Diagram

In this digraph, the circles represent the artifacts created in the software development process and the edges represent a software development activity that creates new artifacts. Later in the dissertation these artifacts and activities are more formally defined as "components" and "steps."

e. Adapting QFD to Software Development

While it is obvious that the production of software differs greatly from the production of hardware and other manufactured products, it is not so obvious what adaptations to the QFD process are required for applying QFD to software development. Since QFD was originally designed to deploy quality in the non-software product industry, it is important to identify those significant differences between software and non-software development and modify the QFD process accordingly. [ZULT90] states that the needed adaptations involve replacing the factors of material and material-based costs by factors of data and time (or schedule). He summarizes these changes as data replacing material, processes replacing functions, and time replacing cost.

(1) Data replaces material. Because software differs from traditional engineering in that it is relatively free of the need for raw material during production, "material" in the QFD process must be replaced with the closest software analog of a raw resource -- "data." Software is unique in product development in that it

can be produced directly from requirements without having to undergo a materials-to-manufacturing process. "Data" becomes the raw resource needed to produce software.

(2) Processes replace functions. In the non-software product industry, "functions" are implemented by "mechanisms" that are made up of "parts." But in software, "functions" are implemented by software "processes."

(3) Time replaces cost. Because software development is not based on a need for raw material, the cost of software development is almost entirely made up of labor costs directly tied to the time required to produce the software product. Thus, "time" replaces "cost" in the QFD matrices.

Another required adaptation of QFD to the software development process is to decide exactly what set of deployment matrices are going to be used and how these matrices are to be linked. There are fairly complicated and complex schemes of matrices that can be used in QFD. [COHE95] and [ZULT92] attribute Akao [AKAO90] with producing the standard advanced book on QFD in which Akao presents what are known as the Akao "Matrix of Matrices". These thirty matrices form the basis of the typical use of QFD in non-software product development. The matrices are summarized in Table 2.

Matrix	"What"	"How"	Activity
A1	Voice of the Customer	Substitute Quality Characteristics (SQC's)	Construct Matrix
A2	Functions	SQC's	Construct Matrix
A3	SQC's	SQC's	Construct Matrix
A4	2 nd level of Design	SQC's	Construct Matrix
B1	Voice of the Customer	Functions	Construct Matrix
B2	Competitive Analysis	Cost	Construct Matrix
B3	Detailed SQCs	Breakthrough targets	Construct Matrix
B4	Critical Parts	SQC's	Construct Matrix
C1	New Technology	1 st Level of Design	Construct Matrix
C2	Functions	1 st Level of Design	Construct Matrix
C3	SQC's	1 st Level of Design	Construct Matrix
C4	2 nd level of Design	1 st Level of Design	Construct Matrix
D1	Voice of the Customer	Product Failure Modes	Construct Matrix
D2	Functions	Product Failure Modes	Construct Matrix

Matrix	"What"	"How"	Activity
D3	SQCs	Product Failure Modes	Construct Matrix
D4	2 nd level of Design	Product Failure Modes	Construct Matrix
E1	Customer Needs	New Concepts	Construct Matrix
E2	Functions	New Concepts	Construct Matrix
E3	SQCs	New Concepts	Construct Matrix
E4	Criteria	New Concepts	Construct Matrix
F1			Value Engineering
F2			Reliability Analysis
F3			Breakthrough Planning
F4			Design Improv. Planning
G1			Quality Assurance Planning
G2			Equipment Deployment
G3			Process Planning
G4			Process Fault Tree Analysis
G5			Failure Mode Effects Analy.
G6			Process Quality Control

Table 2 Akao "Matrix of Matrices" Summary (after [COHE95])

The thirty Akoa QFD matrices are expected to be used as a guide and not to be used verbatim in all development efforts. Since each design team project will have different characteristics, these matrices provide a beginning for teams to create their own QFD Model. [ZULT90] goes further and provides a tailored set of deployment matrices adapted for use in engineering software as illustrated in Figure 13.

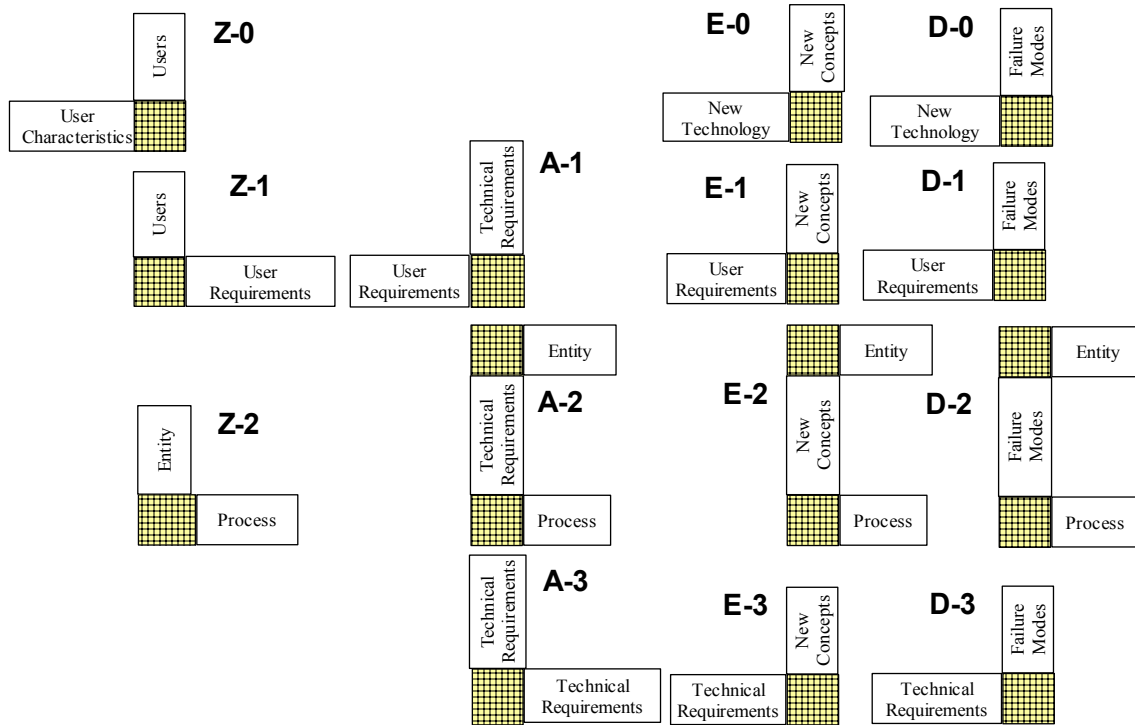


Figure 13 SQFD -- Deployment of the "Customer's Voice" (after [ZULT90])

[ZULT90] has suggested a number of adaptations to Akao's matrix of matrices in accounting for the differences between producing software and producing non-software products. In particular, he adds a series of matrices (the "Z" series) that account for hierarchies and differing perspectives of software stakeholders. The deployment of concerns and viewpoints of the many different types of customers involved in software development is particularly important and requires its own set of matrices [ZULT92]. The software engineer must first understand exactly who all the customers (stakeholders) are, what requirements they have, and to what extent. The requirements analyst must understand the customer needs at a fundamental level before beginning work on the A-1 matrix [ZULT92]. Secondly, he replaces the use of "material" in traditional QFD matrices for "entity" (data) and replaces "function" in traditional matrices with "process". To ensure that the process model (from the data-flow diagram) maps properly to the data model (from the entity-relationship diagram) he proposes the Z-2 matrix. Next, he states that because software, early in its development, is conceptual in nature and can be implemented in many different ways, it is generally not necessary to identify potential conflicts between the same artifacts and he thus, discards the traditional "roof" of the

QFD matrix. If it later proves necessary to perform such a comparison for a particular project, he provides a single "roof-like" matrix -- the A-3 matrix. He points out that new concepts and technology are important ways in which software can exceed customer's expectations and provides the "E" series of matrices to track potential innovation. The "D" series provides a balance between new technology and risk associated with failure modes. Akao's B, C, F, and G series matrices may also be adapted to SQFD.

Just as before, this more complex set of deployment matrices can also be diagramed as a software development process diagram, where each step (edge) is labeled with the corresponding QFD matrix (see Figure 14).

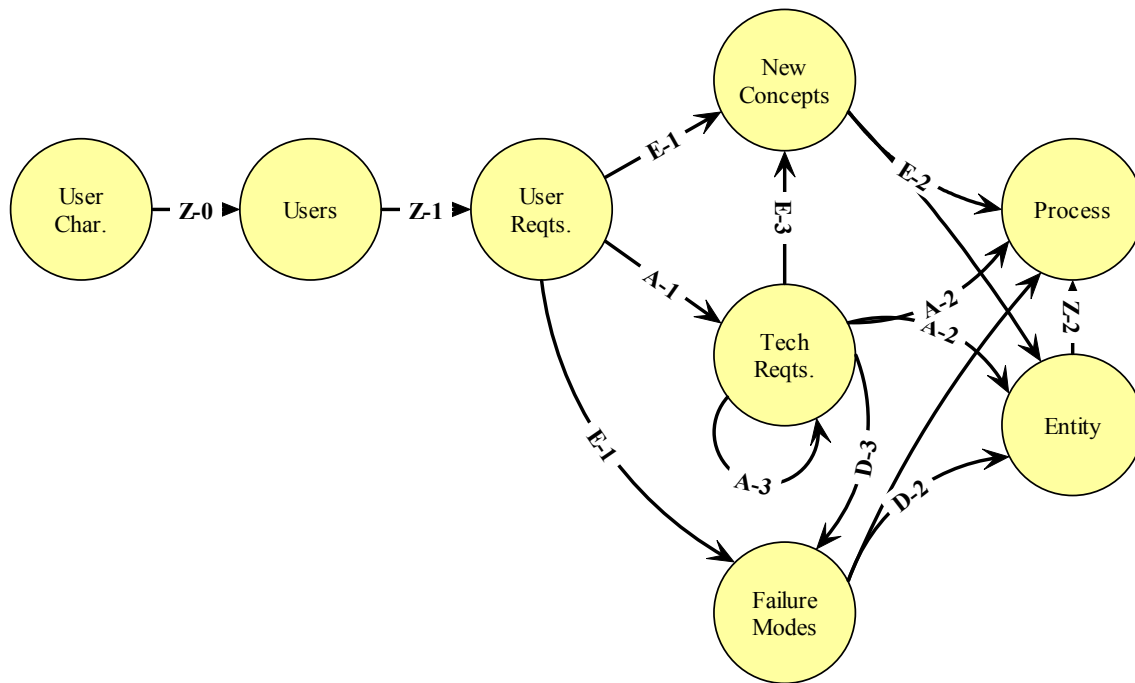


Figure 14 SQFD -- Matrix Deployment as a Process Diagram

Another tailoring of Akao's matrices is performed by [THAC90]. Here the tailoring is designed to account for the planning phases related to the design of embedded software systems (see Figure 15).

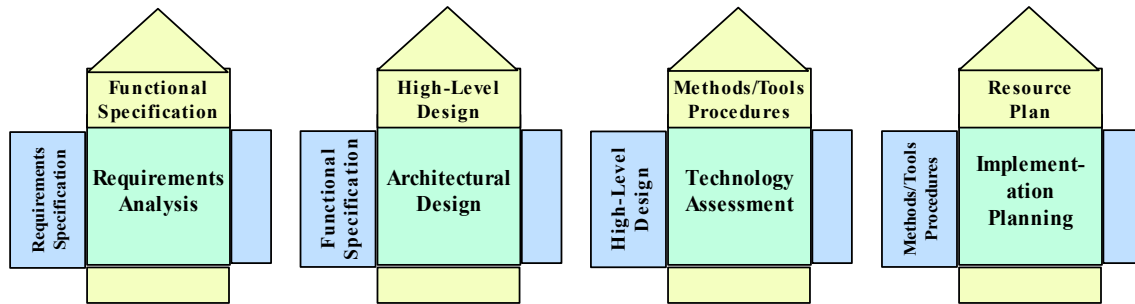


Figure 15 SQFD for Embedded Systems (after [THAC90])

One of the unique aspects of this adaptation is the establishment of a Technology Assessment matrix to provide the ability for companies to analyze the difference between advanced research and product development. Often, a project that began as a research demonstrator or advanced technology prototype becomes the basis for a proposed deliverable product. Unfortunately, the tools needed to actually build the product using the advanced technology do not exist. The Technology Assessment matrix provides a critical test of the viability of the high-level design to ensure that the methods, tools and procedures needed to implement the technology actually exist in the company. If they do not, then it is possible to return to the architectural portion of the design and rework the necessary components [THAC90]. This set of adapted matrices can also be viewed as a software development process diagram (see Figure 16).

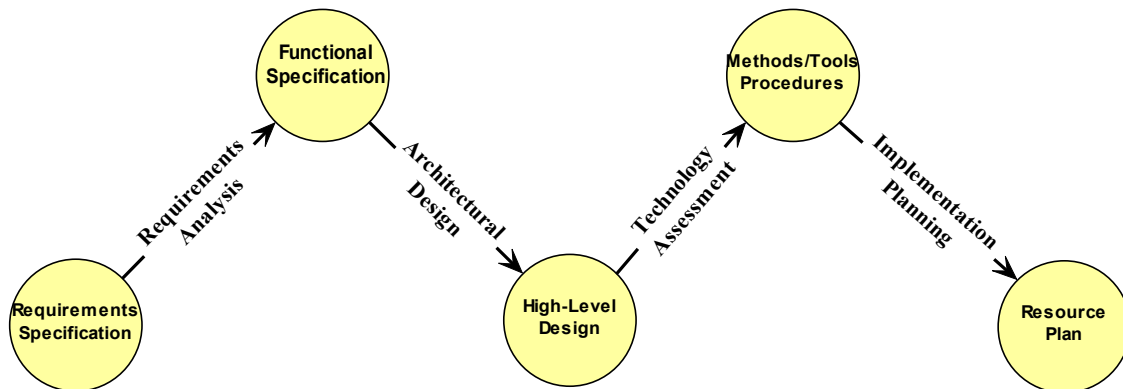


Figure 16 Embedded System SQFD Displayed as a Development Process Model

QFD is often used to implement concurrent engineering, so that parallel portions of the design can be worked on simultaneously. [THAC90] has further adapted a set of deployment matrices for this purpose, where the deployment matrices are used to deliver

key information needed to make a management decision as to whether to proceed with the development effort or not (see Figure 17).

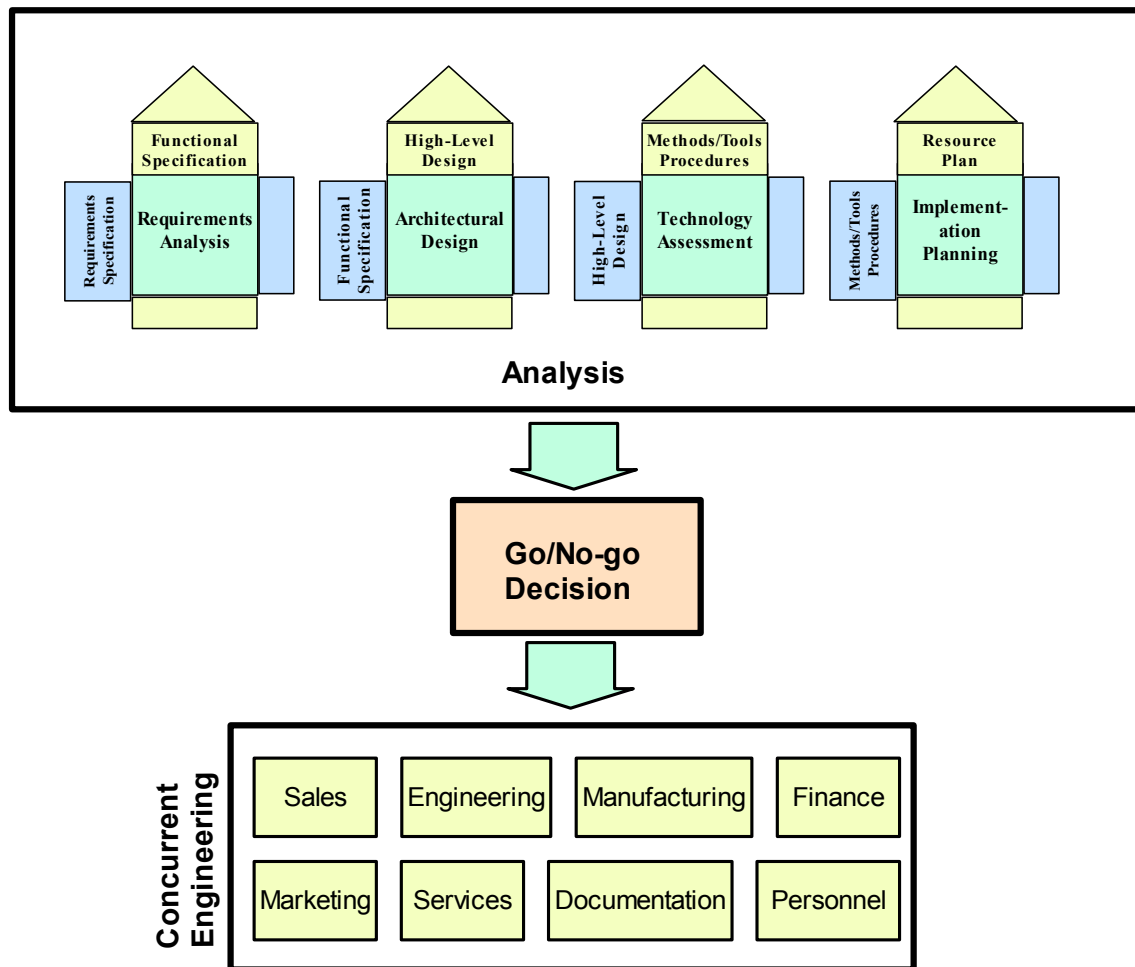


Figure 17 SQFD and Concurrent Engineering (after [THAC90])

What all these various efforts demonstrate is that adapting the QFD process to work with software is possible but requires unique adaptations based on each different development effort used. Thus, as the HFSE seeks to embed QFD within its framework, it will do so in a flexible manner, ensuring that the engineer is able to modify the set of deployments and the entities being linked in the deployments to the engineer's specific project and the engineer's specific software development process.

f. Establishing Correlations

As QFD matrices are formed and linked, one of the key activities becomes establishing the correlation between entities -- correlating the "Whats" to the "Hows."

The use of traditional Japanese symbols is typical in many QFD software packages [COHE95][LIUX00][AKAO90]. [ZULT90, 92] also provides a visual means that intuitively demonstrates the strength of a correlation between two entities. In most cases, these symbols can be typically translated into numerical values of correlation between 0 and 9.

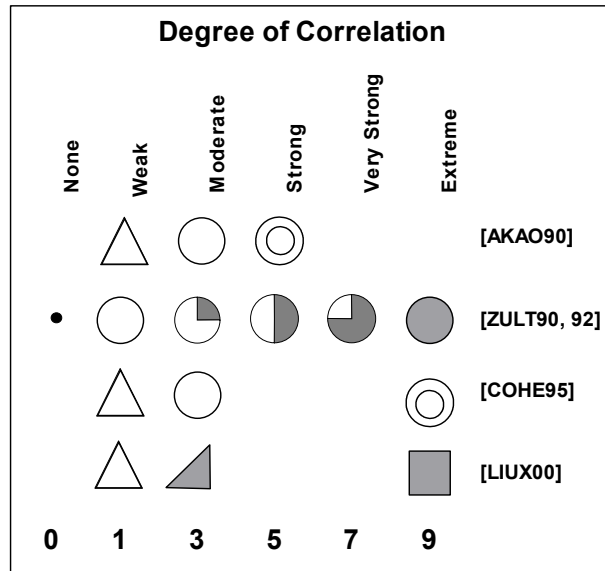


Figure 18 Typical QFD Symbols for Degree of Relationship

Using weight ratios of 1:3:9 helps to highlight the relative importance of software artifacts making interrelations more obvious, even in complex system development [THAC90]. However, as [ZULT92] states, the use of just 1:3:9 limits the accuracy of correlation at the source of input and that additional values (5 and 7) can be used. The engineer has the option of even using finer distinctions if they feel that consistent correlation judgments can be made with precision. [AKAO90] states:

The most troublesome work in quality chart preparation is correlating the demanded quality with quality characteristics [correlating the "Whats" to the "Hows"]. Often, this correlation is based on experience, intuition, and determination... This correlation should be based, however, on knowing and controlling facts. That is, the relationships and their relative strengths should be based on factual data and statistical analysis... reliance on experience, intuition, and determination may be necessary if some or all of the data needed as a basis for confirming the correlating relationships is not available when the quality chart is being prepared. In such cases, we recommend devising some method to differentiate relationships based

solely on experience, intuition, and determination from those based on facts.

[COHE95] asserts that there is no scientific basis for any particular choice of correlation values and that QFD practitioners select the particular scheme that over time has proven to provide them the best separation of important data from noisy data in their application.

When establishing correlation between two entities in a QFD matrix using a 1:3:9 scheme, there are several notes of caution. First, such a scheme assumes a monotonically increasing linear relationship between the two entities. There may, however, be instances where this does not accurately model the relationship. As an example, recall Kano's model of customer satisfaction and requirements (Figure 9); in that model the relationship between customer satisfaction and any particular requirement was first dependent on what type of requirement was being considered (thus a discontinuous relationship) and second was non-linear in cases of "exciting" or "expected" requirements. Finally, the relationship was negative in the case of "expected" requirements (not monotonically increasing). So, how do QFD practitioners handle such complex relationships? In some cases negative correlations have been used in the QFD matrix when a negative relationship exists; but such techniques are often overly complicated and the effort in attaining such precision is often not repaid when the results are presented. The predominate way of handling complex relationships is to attempt to express all entities in such a way that only positive relationships exist, then to make a linear approximation of the relationship and finally to take note of the approximation when latter viewing the results of the matrix [COHE95].

In establishing correlations within the HFSE, the framework will seek to remain flexible in allowing the engineer to use whatever values he deems necessary. The use of positive numerical values will be the norm, as well as an assumption that only positive, linear relationships exist between entities to be correlated.

g. QFD in Large Complex Software Systems

The next issue in embedding QFD into the HFSE is related to scalability, to determine if QFD can be used in large systems with significant complexity. [DEAN92] states that "Complexity has two components: the complexity of the system and the the [sic] complexity of the system to bring forth the system." In other words, the

issue is whether QFD can help to manage the complexity of large systems and/or manage the complexity in the development efforts for large systems. Previous work indicates that QFD provides positive benefits in both areas. In large system design it is useful to map the customer driven quality characteristics against sub-systems of the design. [DEAN92] proposes doing this by using three dimensions of QFD matrix interaction as shown in Figure 19.

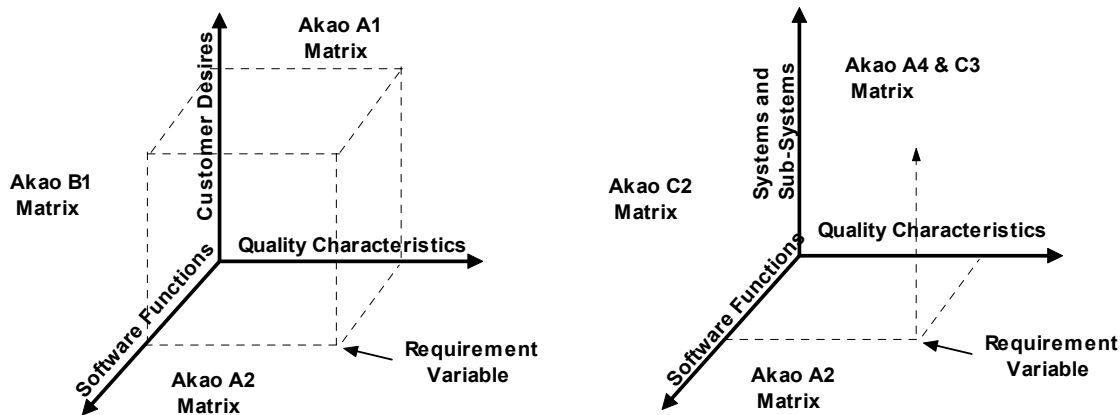


Figure 19 3D View of Matrix Interaction (after [DEAN92])

In this scheme a substitution to account for large systems of systems is to use "subsystem" rather than "part" and to then allocate functions (requirements) to the subsystems. By using deployments to subsystems in large designs, the developers are able to more easily manage the overall complexity of the design. In such cases it is typical to develop matrices in which the interactions between subsystems are defined and correlated.

In addressing the complexity of the development effort itself, QFD can help in assisting with managing the conflicting interests of both customers and developers. In managing customer interests [DEAN92] states that: "Because a large system with NASA has many customers, often with conflicting desires, we found the need to value each customer with respect to the need for the project to satisfy their desires. This quantifies customer political power." In managing the conflicting interests of the many developers (with a wide range of expertise) in large system development it is often better to have generalists (rather than specialists) at the architectural portions of the

QFD interaction. By decentralizing and decomposing the system into meaningful subsystems (with low interaction) it is possible for the specialists then to bring the expertise fully to bear on the problem. "Fortunately, the geometric nature of QFD is a natural medium in which to perform that decomposition and manage the interaction" [DEAN92].

h. The Role of QFD in this Research

As illustrated above, the use of QFD in software engineering has the potential to provide many significant benefits if properly integrated with a software evolution model. Certainly, this integration is one of the central goals of establishing the HFSE. The specific use of QFD in the HFSE can be summarized as follows:

(1) Relationship Tracking. The matrix structure of QFD readily lends itself to establishing traceability of relationships between software development artifacts. The entities tracked in QFD matrices (the "whats" and the "hows") become software artifacts. The correlation matrix becomes the mechanism by which the relationship (and the strength of that relationship) is established and tracked.

(2) Dependency Deployment. While the automated deployment of different kinds of QFD dependencies (different than quality) has been theorized [ZULT93], there has not yet been any formal work in establishing automated mechanisms to make this a reality. The HFSE does exactly that -- it provides a formal description of automated mechanisms that allow software engineers to define specific software dependencies that can then be deployed throughout a software development effort. The engineer then has the ability to select particular views of these dependencies that provide useful decision support information.

(3) Adaptable Matrix Deployment. The many permutations and adaptations to Akao's matrix of matrices dictate that the HFSE must provide a flexible method of allowing an engineer to define their own software development process and their own artifacts. Trying to define a single "one size fits all" set of matrices and artifacts will not properly account for the reality of the many different processes that are used to build software. Additionally, the ability to map a set of matrices to a software development process and vice versa (recall Figure 11 / Figure 12 and Figure 13 / Figure

14) provides a convenient mechanism for abstracting the key underlying relationships represented by the QFD matrices.

(4) Data Linkage. Lessons from previous research indicate that automation is necessary to allow engineers to "browse" through connected design information. The underlying hypergraph structure of the HFSE provides the mechanisms by which this is possible. By linking all the software design data through QFD matrices and then extracting a subset of those matrices will allow the HFSE to provide the engineer useful decision support information.

(5) Cross Communication and Coherency of Design. Embedding QFD into the HFSE will build a framework that enables better cross communication between departments in software development process.

4. Methods for Establishing Interoperability of Software Development Models and Tools

Young points out that consistent representation of the same real world entity in various legacy software products is a continual problem for system interoperability [YOUN01, 02a, 02b]. To address this problem, he presents an Object-Oriented Model for Interoperability (OOMI). This model is used to solve the data and operational inconsistency problems in legacy systems. The model calls for the establishment of a Federation Interoperability Object Model (FIOM) that is specified for a specific group of systems (termed a federation) designated for interoperation. [YOUN01] states:

The FIOM consists of a number of Federation Entities (FEs) that contain the data and operations to be shared between systems. The FIOM also captures the translations required to resolve differences in representation of this data and operations.

An example of an FIOM for a ground combat vehicle is shown in Figure 20.

Federation Interoperability Object Model (FIOM)

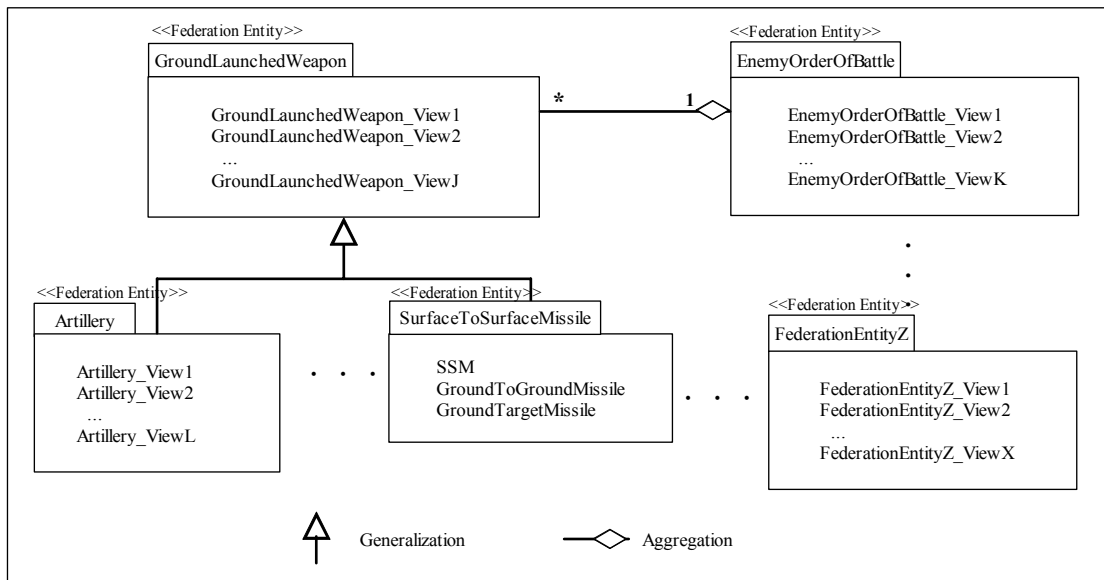


Figure 20 Federation Interoperability Object Model (from [YOUN02a])

At runtime, the OOMI uses a wrapper-based translator to process the information contained in the FIOM. The translator automatically converts instances of real-world entity attributes and operations to the proper representation to enable interoperation between systems. These translations can then be embedded in middleware between target systems as shown in Figure 21.

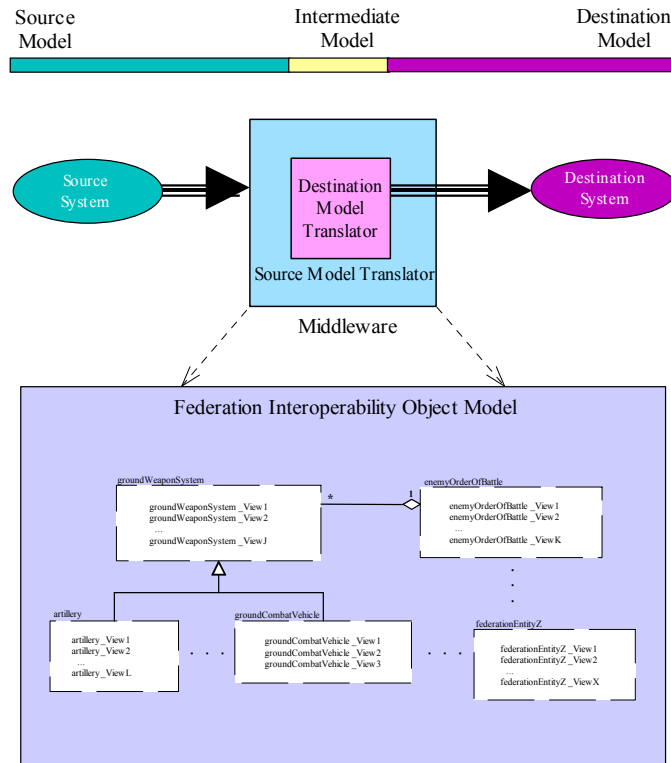


Figure 21 Middleware Based Translation Using the FIOM (from [YOUN02b])

In addition to defining the constructs of the OOMI, Young provides a specialized toolset used to create the FIOM prior to run-time. This tool set is called the Object Oriented Model for Interoperability Integrated Development Environment (OOMI IDE) and is used to accomplish the following:

- Discover the information and operations shared between federation components,
- Provide assistance in identifying the different representations used for such information and operations by component systems,
- Define the transformations required to translate between different representations, and
- Generate system-specific information used to resolve representational differences between component systems.

Young's entire Object-Oriented Model for Interoperability is useful to the dissertation because it provides a mechanism for establishing the interoperability of heterogeneous software development tools and models. The only requirement for these models and tools is that they be definable within an object paradigm.

Young identifies two concepts that are directly applicable to mapping multiple software engineering tools to each other within the HFSE: heterogeneity of scope and heterogeneity of representation. Heterogeneity of scope refers to the fact that differing amounts and types of information can be specified by different systems to represent the state and behavior of the same entity. Heterogeneity of representation refers to the fact that different systems, when referring to the same entity, often have differences in terminology used, format, accuracy, range of values allowed, and structural representation of the included state and behavioral information.

Within the HFSE, attempting to translate behavioral information is particularly challenging. Young points out that "behavioral information can be captured in terms of a set of conditions an element must satisfy or as a set of equations describing the dynamic behavior of the entity" [YOUN01]. Another challenge resolved by the HFSE using Young's methodology was how to resolve different levels of abstraction for information provided in different tools and models. The Federation Entity View (FEV) in Young's OOMI provides the ability to resolve these differences: "The FEV contains the translations required to convert between each component system representation and the 'standard' representation of that view. These translations are used to resolve differences in physical representation, accuracy tolerances, range of values allowed, and terminology used in representing a federation entity view. These translations are defined by the interoperability engineer and stored in the FEV for subsequent use" [YOUN01].

Another challenge relates to communication within the FIOM. Young assumes a publish/subscribe mechanism that assumes a one-directional broadcast of information from either information sources or information consumers. Updates are a new issue in the context of software development environments - more than one node (designer, stakeholder, etc.) can propose modifications to the same real world object (design component refinement, etc). This may need more sophisticated methods to maintain consistence.

5. Application of Ontologies for Interoperability

a. Ontology Overview and Example

The term "Ontology" is borrowed from philosophy where it is defined as a systematic investigation of "Existence". The term is now widely used in Artificial

Intelligence and Knowledge Engineering where what "exists" are those entities which can be "represented." Ontology is the term used to refer to the shared understanding of some domain of interest that may be used as a unifying framework to solve problems in that domain [USCH96]. An ontology necessarily entails or embodies a world view with respect to a given domain. This world view is often conceived as a set of concepts (e.g. entities, attributes, processes) along with their definitions and their inter-relationships. Because people, organizations, and software systems must communicate between and among themselves, there are often difficulties/inaccuracies in communications because of differing contexts, understandings, viewpoints and assumptions. Therefore, ontologies help to accomplish the following:

- Improve poor communication,
- Establish a unifying framework for conceptual models and ideas,
- Establish the basis for interoperability, and
- Prevent redundant work and cross purposes.

[GRUB95] defines an ontology more formally as "an explicit specification of a conceptualization" where a conceptualization is "an abstract, simplified view of the world that we wish to represent for some purpose" and consists of "objects, concepts, and other entities that are assumed to exist in some area of interest and the relationships that hold among them." The widespread use of ontologies provides meaningful mechanisms for distinguishing various types of objects (concrete and abstract, existent and non-existent, real and ideal, independent and dependent) and their ties (relations, dependences and logic). Another formal definition is offered by Sowa [SOWA00] as quoted by [LENC01] who states that an ontology is:

a catalogue of the type of things that are assumed to exist in a domain of interest D , from the perspective of a person who uses a language L for the purpose of talking about D .

[LENC01] adds:

From a semantic point of view, an ontology determines the domain of discourse for a language L , i.e. what L talks about. The ontology on which L is interpreted actually constrains the expressiveness of L itself. For instance, if the ontology only contains plants and animals, then it will be impossible to speak about computers, unless they are categorized either as plants or as animals, thereby losing the possibility to account for crucial

differences among them. To be able to do this, the ontology should be refined by adding a further category, e.g. the one of artifactual [sic] objects.

Ontologies reduce or eliminate conceptual and terminological confusion. They establish a shared understanding and unifying framework which improves communication, consistency and ambiguity, integration of differing perspectives, interoperability, and systems engineering [USCH96].

(1) Communication. Ontologies help to improve communication between people with different needs and viewpoints arising from differing contexts. Examples include normative models that establish the semantics of the system and potential extensions as well as networks of relationships that explore the relationships between entities.

(2) Consistency and Ambiguity. Depending on the level of formality used in its establishment, an ontology provides explicit definitions within the domain, eliminating ambiguity and providing consistent interpretation across the domain.

(3) Integration of Differing Perspectives. Ontologies provide a framework for integrating different user perspectives. Often the commonalities in the differing user perspectives can form the groundwork for development of standards within the community. The differences in the perspectives often lead to different ways of categorizing information within the ontology (e.g. in an ontology about plants, with differing perspectives of farmers and florists, different sets of information about each plant would be needed to support both perspectives).

(4) Interoperability. Interoperability among systems is achieved by translating between different modeling methods, paradigms, languages, and software tools. Examples include the following:

- Integration environment for tools,
- Inter-lingua translators,
- Internal interoperability: integration of systems (perhaps legacy systems), and
- External interoperability: insulation of the organization from the outside world.

(5) System Engineering. Ontologies support improvements in system engineering (e.g. reuse, reliability, specification). Examples include the following:

- Specification. Ontologies provide a shared understanding that assists in establishing the specifications of systems and prevents later misinterpretation of those specifications.
- Reliability. Ontologies can form the basis for manual and automated verification. Formal ontologies can be used to make assumptions explicit to users.
- Reusability. Common and explicit understanding allows modules to be imported and exported between systems.

The formalism used in specifying ontologies varies widely. Ontologies can range from being very informal to very formal [USCH96]. Consider the following formalisms:

- Highly Informal: loosely expressed in natural language,
- Semi-Informal: expressed in restricted and structured form a natural language,
- Semi-Formal: expressed in an artificial formally defined language, and
- Rigorously Formal: meticulously defined terms with formal semantics, theorems and proofs of such properties as soundness and completeness.

As an example of an ontology, the Enterprise Ontology [USCH98] was developed within the Enterprise Project (a collaborative effort between the Artificial Intelligence Applications Institute (AIAI) at the University of Edinburgh, IBM, Lloyd's Register, Logica UK Limited, and Unilever). The Enterprise Ontology provides a framework for enterprise business modeling and is presented as a collection of terms and definitions relevant to business enterprises. [USCH98] uses natural language definitions for all the terms and is an example of a semi-informal ontology. Table 3 lists the terms defined in the Enterprise Ontology.

Major Category	Ontology Terms
Activity	Activity, Activity Specification, Execute, Executed Activity Specification, T-Begin, T-End, Pre-Condition, Effect, Doer, Sub-Activity, Authority, Activity Owner, Event, Plan, Sub-Plan, Planning, Process Specification, Capability, Skill, Resource, Resource Allocation, Resource Substitute
Organization	Person, Machine, Corporation, Partnership, Partner, Legal Entity, Organizational Unit, Manage, Delegate, Management Link, Legal Ownership, Non-Legal Ownership, Ownership, Owner, Asset, Stakeholder, Employment Contract, Share, Share Holder
Strategy	Purpose, Hold Purpose, Intended Purpose, Strategic Purpose, Objective, Vision, Mission, Goal, Help Achieve, Strategy, Strategic Planning, Strategic Action, Decision, Assumption, Critical Assumption, Non-Critical Assumption, Influence Factor, Critical Influence Factor, Non-Critical Influence Factor, Critical Success Factor, Risk
Marketing	Sale, Potential Sale, For Sale, Sale Offer, Vendor, Actual Customer, Potential Customer, Customer, Reseller, Product, Asking Price, Sale Price, Market, Segmentation Variable, Market Segment, Market Research, Brand, Image, Feature, Need, Market Need, Promotion, Competitor
Time	Time Line, Time Interval, Time Point

Table 3 Overview of the Enterprise Ontology (after [USCH98]).

The table illustrates a collection of terms and definitions relevant to business enterprises. This collection is presented in natural language and is classified into major categories. An example of the definition of the term "Corporation" under the major category of "Organization" is as follows [USCH98]:

CORPORATION: a group of PERSONS recognised in law as having existence, rights, and duties distinct from those of the individual PERSONS who from time to time comprise the group.

Notes:

1. Historically, in law, rights and duties apply to individual humans; rights and duties of groups are inherited from this.

Note that the definition of CORPORATION uses other terms from the ontology (i.e. PERSON) and thus establishes a relationship between these two terms. The definition also includes additional annotations (notes) that further clarify the definition, reducing ambiguity.

Ontologies play an important role in the interoperability aspects of the HFSE and the concepts about ontologies in this section lay the foundation for this role. As a basis of implementing Young's OOMI, the interoperability engineer must begin with an ontology of the domain. [YOUN02b] illustrates this essential role in FIOM Construction Use Case in Figure 22.

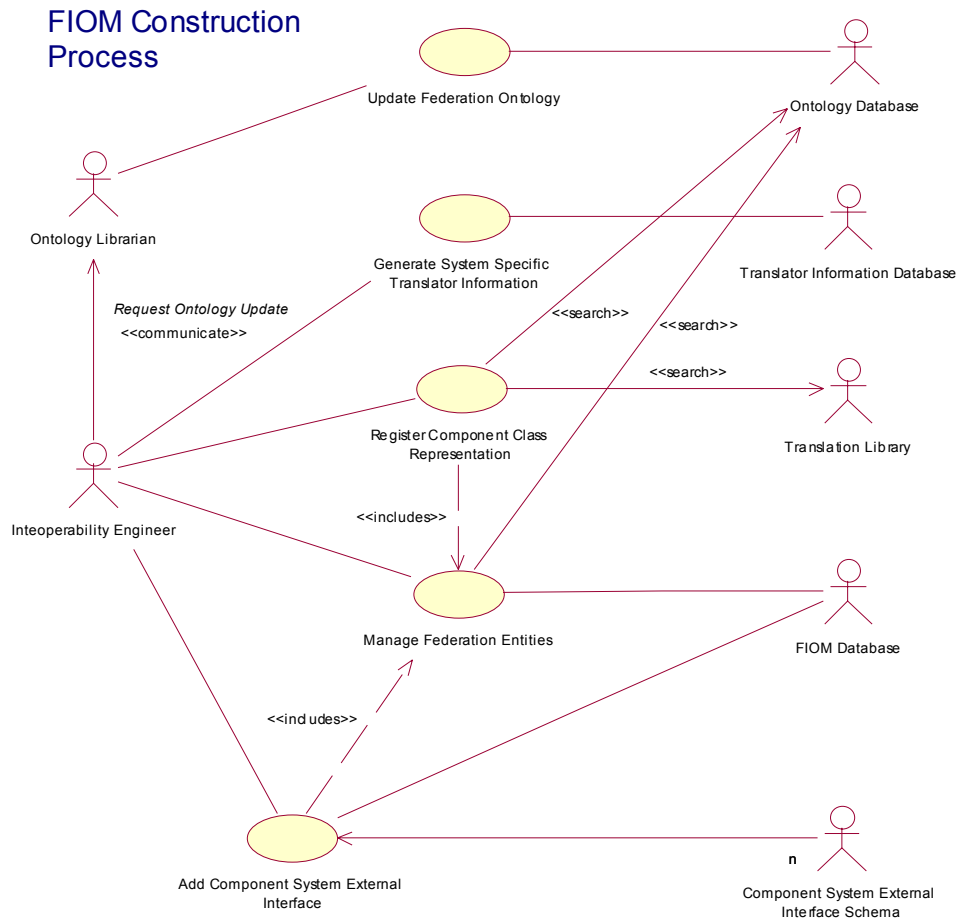


Figure 22 Role of Ontology in FIOM Construction (from [YOUN02b])

In the case of the HFSE, the domain is that of software development tools (and artifacts produced by those tools). Thus, a first step in establishing the HFSE is to construct an ontology for the set of tools to be integrated by the HFSE. The majority of the work in establishing this software development tool ontology was undertaken by [HASN03] in support of this dissertation.

b. Constructing an Ontology

There is no current field of "ontological engineering" comparable to the field of knowledge engineering, so there are no standard accepted methodologies for the building of an ontology. [USCH96] provides a general methodology for constructing an ontology that can be modified and fit to particular purposes and consists of the following steps:

- Identify the purpose and scope of the ontology,
- Build the ontology,
 - Capturing the ontology,
 - Coding the ontology,
 - Integrating existing ontologies,
- Evaluate the ontology, and
- Document the ontology.

This general methodology consists of four main steps detailed below.

(1) Identify the Purpose and Scope of the Ontology. One of the most important steps in constructing an ontology is to make an early decision as to the purpose of the ontology. This purpose provides a controlling perspective on the terms, attributes of terms, and relationships captured in the ontology. The scope of the ontology provides a guide to the depth and breadth of the intended ontology, consistent with the purpose. For instance, if the ontology is to be used for interoperability, then the engineer must establish both semantic and syntactic detail (at an appropriate level of detail) in order to ensure that information can be transmitted and used between systems.

(2) Build the Ontology. The first step in building the ontology is to capture the key concepts and relationships in the domain and then to precisely and unambiguously define these terms. In coding the ontology, the goal is to develop an explicit representation of the domain using the concepts and relationships already captured. This "coding" can be performed with increasing levels of formality. The engineer must decide upon a representation language and must decide upon the meta-data that will be used to formally express each ontology term and relationship. Another consideration is to decide to what degree (if any) should already existing ontologies be integrated into the new ontology. This of course needs to be consistent with the scope of the ontology agreed previously.

(3) Evaluate the ontology. Specific evaluation criteria for the ontology should be established. Such criteria could be based on the general guidelines [GRUB95] provides for constructing an ontology. These guidelines include clarity, coherence, extensibility, minimal ontological commitment, and minimal encoding bias. The ontology is then iteratively compared to these criteria and refinements are made as needed. [GRUN95] offers a more formal approach to the evaluation of ontologies using formal competency questions and completeness theorems. His approach requires a formal ontology in which the ontology definitions and constraints on their interpretation have been defined using first-order logic.

(4) Document the ontology. The final step is to document the ontology. All assumptions about the domain should be annotated as well as information about the meta-data used to describe the ontology. Of particular importance is to document the ontology boundaries (consistent with the scope). This documentation can take the form of textual descriptions, formal predicates, and UML diagrams.

[NOYN01] offers a slightly more specific set of steps for constructing an ontology; however, her methodology assumes the use of the Protégé ontology capture tool [PROT03a]. The seven steps in her methodology include the following:

- Determine the domain and scope of the ontology (similar to (1) above),
- Consider reusing existing ontologies (as in step (2) above),
- Enumerate important terms in the ontology (as in step (2) above),
- Define the classes and the class hierarchy (using Protégé),
- Define the properties of classes -- slots (using Protégé),
- Define the facets (types of properties) (using Protégé), and
- Create instances of the classes (using Protégé).

This methodology does not address step (3) (ontology evaluation) of the [USCH96] methodology; but, it does provide more detail and a specific tool for accomplishing steps (2) and (4).

The general methodology proposed by [USCH96] and the more specific methodology by [NOYN01] are important to this dissertation in that they form the template by which a methodology is established for building the software development tool ontology upon which the HFSE relies. The details of this modified methodology (and the results from the effort) are presented in Chapter III.

*c. **Ontology Definition and Capture***

Steps (2) and (4) in the general ontology design methodology above imply that some of the key activities involved in ontology design center around building and documenting the ontology. In the case of this dissertation, the tool used for both of these steps is an ontology capture tool developed at Stanford University called Protégé-2000. This tool can be used to define the meta-data, the structure of the information of the ontology, as well as to capture specific instances of ontology classes (consistent with that structure).

Protégé-2000 is a knowledge-based design and knowledge-acquisition system. It is available free from Stanford University's Protégé project homepage [PROT03a] and is compatible with a wide range of knowledge representation languages [PROT03b]. The tool allows the ontology designer to create custom knowledge-based tools for particular applications. Protégé assists software developers in creating and maintaining explicit domain models, and in incorporating those models directly into program code. The core concept behind the architectural makeup of Protégé-2000 is ontology design. The tool allows the designer to establish the granularity of the design in a domain-specific area. Then using problem-solving methods specific to that domain, domain experts can then search the ontology knowledge base.

The Protégé knowledge tool uses four main concepts that are represented in the software by frames. These are

- Classes,
- Instances,
- Slots, and
- Facets.

Classes represent the definitions of concepts; instances represent the specific examples of a concept; and slots represent attributes of either a class or an instance. Finally there are facets, which are defined as properties of slots, and are constraints on slot values [PROT03a]. Figure 23 is screen shot of the Protégé-2000 knowledge acquisition tool.

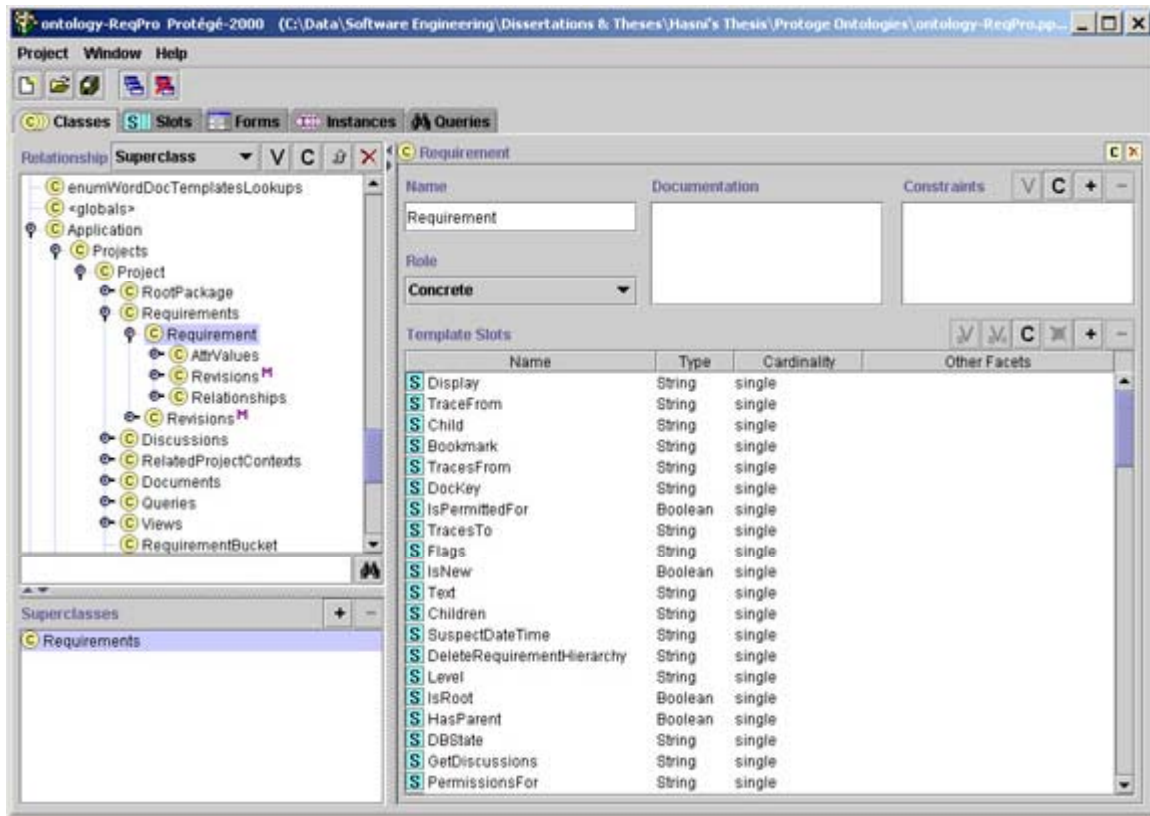


Figure 23 Protégé Screen Shot

This particular screen shot is taken from the Requisite®Pro Ontology (one of the subordinate ontologies of the HFSE presented later in the dissertation). Note the use of a class-hierarchy tree in the upper left panel, with slots for the selected class displayed in the lower right panel. Protégé offers tabs for navigating between classes, slots, forms (to collect data), instances, and queries (to collect specific data from the ontology database).

While Protégé with its hierarchical class structure at first looks much the same as an object-oriented software approach, there are differences. [MUSE98] and [NOYN01] both point out that ontology development using Protégé is different from that taken in traditional object-oriented programming. In object-oriented approaches both the domain knowledge (the attributes of objects) and the problem solvers (the methods) are bundled together. Sending messages from one object to another controls program execution. Each object encapsulates both data and the methods that operate on that data. In the Protégé approach, however, the problem-solving methods are separate entities unto themselves and have formal parameters that must be mapped to the appropriate classes in

the domain knowledge (i.e. the methods (Protégé slots) have an existence outside of and distinct from the class).

The Protégé tool itself is GUI-based; all the design is done using forms and tabs. The tool also employs a visualization tool that allows the designer to see and edit the ontology structure. The Protégé API provides designers the ability to add plug-ins and to access domain knowledge stored in the Protégé tool from other applications.

The role of Protégé in the dissertation is that it is the ontology definition and capture tool that is used to define the software development tool ontology required by Young's OOMI for establishing interoperability between heterogeneous software development tools. The ontology data-base required by [YOUN02b] (recall Figure 22) is that of Protégé. A summary of the HFSE ontology results from Protégé are presented in Chapter III; however, for a more complete and detailed explanation of how Protégé was used to support this dissertation, see [HASN03].

d. UML as an Ontology Description Language

[CRAN99, 01] and [KOGU02] present the Unified Modeling Language as a possible language for defining and describing domain ontologies. Object-Oriented Modeling (OOM) and the Unified Modeling Language (UML) have established a significant following in the field of software engineering. Because of this acceptance and the fact that OOM and UML are widely supported by robust commercial tools, the use of UML for ontology representation is attractive. [KOGU02] offers the following additional reasons why an ontology designer should consider UML as a representation language for ontologies:

- UML is graphical and easily understood.
- UML is an open standard managed by the OMG.
- UML has standard mechanisms for defining extensions.
- Real world systems often have existing UML models.

[CRAN99] points out that UML by itself is often not expressive enough to explicitly define ontology terms and constraints. He points to the use of the Object Constraint Language (OCL) in refining relationships on and between classes as a way of overcoming this shortcoming.

Because Young's interoperability model is highly reliant on an object structure within the FIOM, the use of UML to define the relationships between ontologies in the HFSE makes sense. The OOMI methodology uses a UML type structure to express the inter-relationships between objects in different ontologies – mirroring that implementation will make it easier to apply Young's methodology [YOUN02b].

C. RELATED WORK

The foundations for the contributions of this dissertation were laid in the previous section. In this section, the focus is on identifying the related work by others (that work which attempts to achieve a similar purpose as the HFSE). This section identifies and distinguishes how this related work differs from that accomplished by this dissertation.

1. Software Development Tool Suites: The Rational Approach

While there are many software development tool suites, perhaps the most significant work to date in developing a large integrated set of powerful tools for building software has been undertaken by Rational Software Corporation.

a. Summary

[KRUC96] provides an overview of the Rational Development Process. He describes the Rational Development Process for software as a highly automated, object-oriented, iterative and incremental software development process. It is centered around three main pillars: people, process, and tools/methods. The process can be approached from two main perspectives: management and technical. From the management perspective there are four main phases of development: inception, elaboration, construction, and transition. From the technical perspective, development is best viewed as a series of incremental iterations concluding with the release of a product. The two perspectives synchronize through the production of artifacts related to development as illustrated in Figure 24 below.

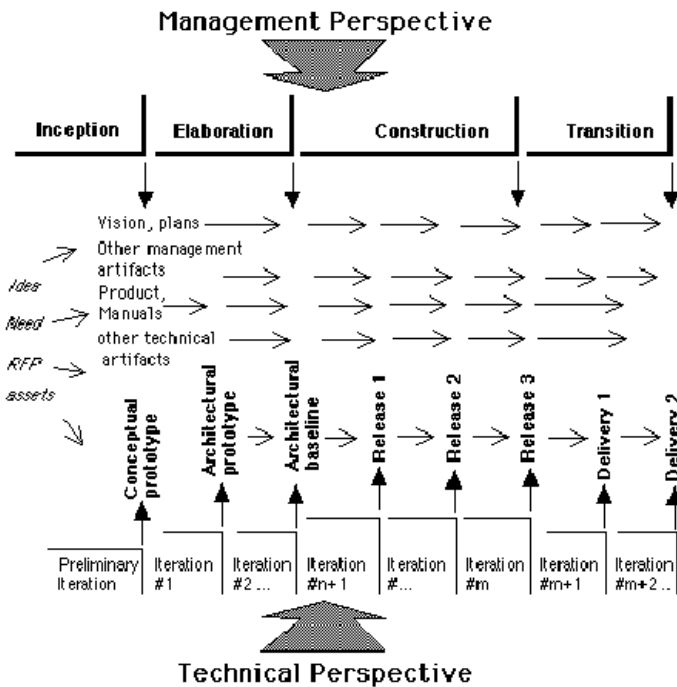


Figure 24 Synchronization of Perspectives in the Rational Process (from [KRUC96])

The Rational process identifies a number of artifacts; however, the software itself is not considered an artifact. These artifacts include the following:

- Management artifacts: organizational policy, vision, business case, development plan, evaluation criteria, release description, deployment document, status assessment; and
- Technical artifacts: user's manual, software documentation, software architecture

Intellectual activity (such as planning, analysis, design, etc.) can be done in any phase. Figure 25 gives an example of the amount of intellectual activity that might take place during any particular phase.

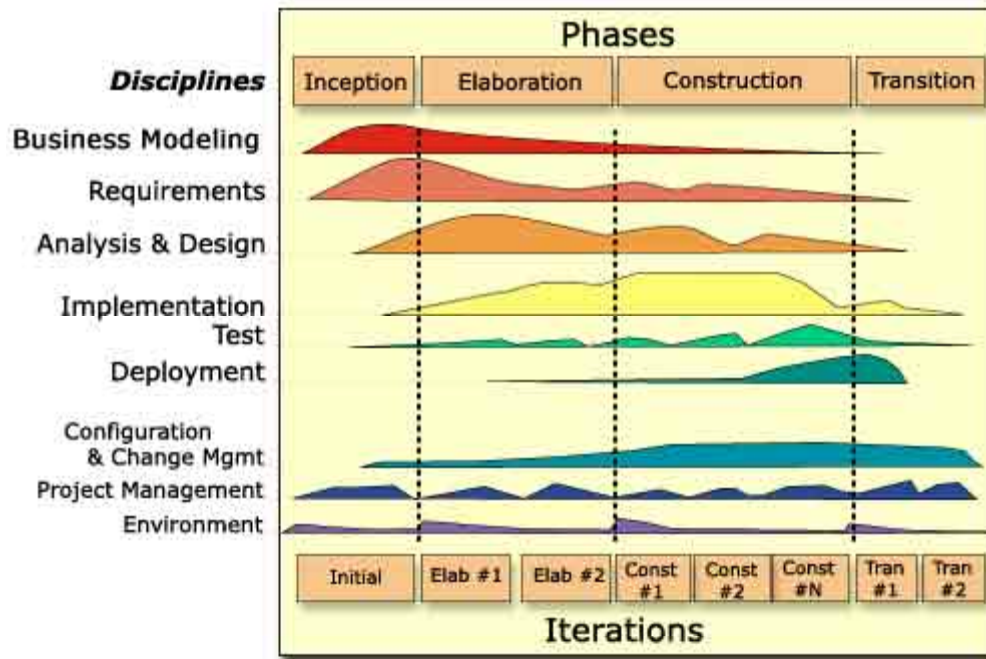


Figure 25 Intellectual Activity in the Rational Process (from [RATI03])

While continually growing, the Rational's tool support for software development is substantial. [RATI98] points out that any software engineering process requires tools to support development activities throughout the software's lifecycle:

An iterative development process puts special requirements on the tool set you use, such as better integration among tools and round-trip engineering between models and code. You also need tools to keep track of changes, to support requirements traceability, to automate documentation, as well as tools to automate tests to facilitate regression test. The Rational Unified Process can be used with a variety of tools, either from Rational or other vendors. However, Rational provides many well-integrated tools that efficiently support the Rational Unified Process.

The following are the automated tools available from the Rational suite to support software development:

- **Rational Requisite®Pro** -- a requirements engineering tool that makes requirements easy to write, communicate and change.
- **Rational ClearQuest™** -- a change-request management product that enables project teams to track and manage all change activities.
- **Rational Rose®** -- a visual modeling tool for business process modeling, requirements analysis, and component architecture design.

- **Rational SoDA®** -- automates the production of documentation for the entire software development process.
- **Rational Purify®** -- a run-time error checking tool for application and component software developers programming in C/C++.
- **Rational Visual Quantify™** -- a performance profiling tool for application and component software developers programming in C++, Visual Basic, and Java.
- **Rational Visual PureCoverage™** -- identifies areas of code not exercised in testing so developers can thoroughly, efficiently and effectively test their applications.
- **Rational TeamTest** -- creates, maintains and executes automated functional tests, allowing the test team to thoroughly test the code and determine if the software meets requirements and performs as expected.
- **Rational PerformanceStudio™** -- a tool that measures and predicts the performance of client/server and Web systems.
- **Rational ClearCase®** -- a software configuration management tool.

These tools have been specifically designed to be interoperable and to function by using a "model" approach. The idea is that a model of the software system is created and specific artifacts related to the model are produced by the tools.

b. Relationship to the Dissertation Topic

There are several concepts from Rational's integrated tools suite that are related to work presented in this dissertation.

(1) Management and Technical Perspectives. The Rational Development Process has both management and technical perspectives, as does the HFSE. These perspectives allow particular stakeholders to glean particular information from the development effort subject to their needs.

(2) Process Dependence. The Rational Development Process ties together people and tools through process (and procedures and artifacts). The tools supplied by Rational are integral to the Rational Unified Process. The HFSE, on the other hand, ties together tools for people, independent of process.

(3) Diversity of Artifacts. The diversity and number of management and technical artifacts imply a need for life cycle coverage. Because the suite is managed and maintained by a large software developer with a significant user base, artifacts developed today will likely be able to be viewed decades later (as long as

Rational remains in business). This may not necessary be the case in the HFSE, where artifacts developed with tools that the user has today may not be available (because of software/hardware obsolescence) decades later. This presents an issue that tests long-term openness/extensibility of the HFSE and is an issue to be considered in future research.

c. Weaknesses

The main weakness of Rational's tool suite is one of "Legacy System Interoperability." Rational Software Corporation's history is one of software development tool acquisition -- collecting the "killer apps" of the software development domain. Rational then lashed together these tools and developed a "unified" process around the tools rather than define the process, then develop tools that support that process. This is not to say that what they have accomplished is not without merit. In fact, the popularity of Rational's tool suite speaks volumes about its usefulness to produce real software products. However, in the ideal world, process should come first, with tools specifically designed and tailored to support the process -- not the other way around. Otherwise, important aspects of the process could be left out or forgotten simply because they were not supported by the tools available.

The main advantage of the approach taken with development of the HFSE over the Rational tool approach is that the HFSE is specifically designed to account for legacy software development tools and processes. Software engineers construct the HFSE around their already existing software development process and tools. The HFSE allows software designers to interoperate between the tools that they already use rather than have to rely on tailoring an integrated tool suite to their use. They will be able to use the process they want with the tools they want -- building up their development environment over time.

2. Rational Unified Process (RUP)

As in the case of software development tool suites, there are numerous software development process models that provide software engineers methodologies for producing software. To some degree, each of these process models attempts to provide some of the same aims as the HFSE; namely, providing software engineers an ability to have life-time visibility and leverage of all the software development artifacts produced.

These process models include the waterfall model, the linear sequential model, the prototyping model, the rapid application development model, the evolutionary model, the spiral model, the Win-Win model, formal methods, etc. A good survey of each of these process models is provided in [PRES01] and [SOMM01]. Rather than attempt to compare each of these models against the HFSE, the focus of this section will be to examine just one process model (Rational's Unified Process (RUP)) and contrast it against the HFSE approach.

a. Summary

[RATI98] is a guide to using the Rational Unified Process® and was produced by Rational Software Corporation to provide software developers a set of six best practices to employ when using the process. [RATI98] begins with a summary of the RUP:

The Rational Unified Process® is a **Software Engineering Process**. It provides a disciplined approach to assigning tasks and responsibilities within a development organization. Its goal is to ensure the production of high-quality software that meets the needs of its end-users, within a predictable schedule and budget... The Rational Unified Process is a **process product**, developed and maintained by Rational® Software... The Rational Unified Process enhances **team productivity**, by providing every team member with easy access to a knowledge base with guidelines, templates and tool mentors for all critical development activities... The Rational Unified Process activities create and maintain **models**. Rather than focusing on the production of large amount of paper documents, the Unified Process emphasizes the development and maintenance of **models**—semantically rich representations of the software system under development... The Rational Unified Process is supported by **tools**, which automate large parts of the process. They are used to create and maintain the various artifacts -- models in particular -- of the software engineering process: visual modeling, programming, testing, etc. They are invaluable in supporting all the bookkeeping associated with the change management as well as the configuration management that accompanies each iteration... The Rational Unified Process is a **configurable process**. No single process is suitable for all software development. The Unified Process fits small development teams as well as large development organizations. The Unified Process is founded on a simple and clear process architecture that provides commonality across a family of processes.

The RUP provides a great deal of automation for many aspects of software development. The RUP relies on unifying process, people, and tools into a management and technical perspective as shown in Figure 26 [KRUC96].

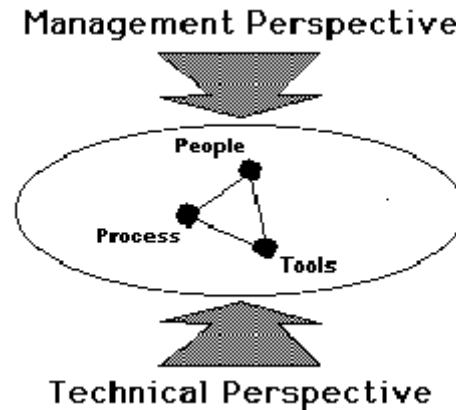


Figure 26 Rational Unified Process (from [KRUC96])

In addition to providing an overview of the RUP, [RATI98] identifies and describes six best practices to employ when using the RUP.

(1) Develop software iteratively. Apply an approach that takes advantage of an increasing understanding of the problem during iterative refinements.

(2) Manage requirements. Elicit, organize, and document the system's required functionality and constraints. Document design decisions and tradeoffs.

(3) Use Component-Based Architectures. Establish an early baseline architecture that is flexible, understandable, and promotes software reuse.

(4) Visually Model Software. Capture the structure and behavior of architectures and components while hiding their details. Use graphical building blocks and abstraction to convey main ideas.

(5) Verify Software Quality. Review the software with respect to the requirements. Check reliability, functionality, application performance, and system performance.

(6) Control Changes to Software. Change is inevitable. The successful project requires that you control, track, and monitor changes.

b. Applicability or Relationship of Work to the Dissertation Topic

There are two concepts associated with the RUP that are similar to the approach taken by the HFSE.

(1) The idea that the Rational Process is configurable is noteworthy. The HFSE is configurable as well; however, as Rational provides a suite of tools where a software development organization "turns-off" that functionality which it does not use or is not applicable; users of the HFSE will "build-up" their configuration by adding only those tools which are useful.

(2) Rational uses Tool Mentors to provide users a step-by-step guide describing in detail how to operate a tool, (i.e. what menus to launch, what information to enter into dialog boxes, and how to navigate a tool) to carry out an activity within the process. The Mentors allow users to link the tool-independent process to the actual manipulation of the tools. The main idea here useful to the dissertation is that Rational still requires the software engineer to be the primary linkage between tools. The tools themselves contain some compatible file formats and processes, they allow cut and pasting between tools, but there is not an automated linkage between all relevant objects between all the tools.

c. Weaknesses

The Rational Unified Process differs from the HFSE approach in the following four significant ways:

- Over-reliance on people and process as unifying factors,
- Use of only specific proprietary tools,
- Configurability of the tool set, and
- Lack of a QFD style dependency metric that links artifacts in all phases of software development.

While the Rational development process unifies specific tools through people and process (and procedures and artifacts), the HFSE unifies any tools for people, independent of process. In other words, the Rational process is overly reliant on people and process to be the unifying factor between their specific set of tools. The HFSE is not dependent on any specific set of tools and instead will allow the developers to unify the tools with which they are most familiar. Users of the HFSE will "build-up" their configuration by adding only those tools which are useful. In other words, the Rational

Process is configurable only as long as software engineering is restricted to subsets of the tools provided by Rational (and a few Microsoft office tools for which they have accounted); while, the HFSE will accommodate an open universe, including tools that have not been developed yet. Finally, the Rational approach does not provide for the QFD style of holistic linkage of all artifacts created in the software development process. Lacking this linkage, it is impossible to gain adequate visibility of the effects of particular metrics on other phases of the development process.

3. Integrated Software Development Environments

There has been a significant amount of software engineering research associated with Integrated Software Development Environments (ISDEs), Integrated Project Support Environments (IPSEs), and Integrated CASE tools (I-CASE). This includes a workshop series (ACM SIGSOFT's Software Engineering Symposium on Practical Software Development Environments (SESPSDE) over a 5 year period in the late 1980's and early 1990's) as well as a major DARPA program (called Arcadia). This previous work differs from the HFSE primarily in that the HFSE focuses on the holistic nature software artifacts and the relationships between them (captured by QFD) whereas these other previous efforts focused mainly on electronic syntactic data interoperability between tools. These previous efforts cast the HFSE as unique in terms of interoperability, but not particularly original. However, the integration of QFD into the RH model to provide rich dependency relationships between development artifacts remains both unique and original.

a. ISDEs and IPSEs

[BROW92, 93] examines and summarizes some of the problems in developing ISDE/IPSE technology. He points out that the research associated with software development tool support has centered around two main approaches.

(1) IPSE Approach. In this approach, the IPSE developer attempts to provide an infrastructure for common services required by multiple tools. The environment is often constructed around a particular software development process and provides specific services in support of that process. [BROW93] states: "The majority of the work in this area has explored the common services that need to be

provided by such a framework, and the consequent interaction between the framework and the tools which are embedded within it.”

(2) Computer-Aided Software Engineering (CASE) Tool Approach. In this approach, market needs and customer feedback drive the expansion of existing CASE tools to encompass ever-greater functionality and provide additional features to support ever-increasing portions of the software development process.

It is the IPSE approach that most closely parallels the approach in the HFSE, so it is useful to examine why the IPSE approach has yet to meet expectations for improving software development. [BROW93] cites a number of reasons. First, there is little economic evidence that IPSE approaches are effective and without such evidence developers are reluctant to adopt such technology. Developers would much rather purchase individual CASE tools and glue them together. [BROW93] states:

...the purchase of a CASE tools is *not* seen as a strategic decision, but more as a pragmatic one. For example, it is often found to be easier to obtain money and management support for purchasing a new CASE tool, or integrating a set of CASES tools, than for investing in IPSE technology. This is due to the incremental nature of the investment, the more visible improvements in productivity they often bring, and the more manageable complexity of the new software.

Second, IPSEs’ size and complexity makes them unmanageable; often there is a perception that productivity will actually decrease because of this complexity. Third, the lack of flexibility and generality in IPSE approaches makes them less attractive. While generality is a goal of most IPSE technologies, few in reality are flexible enough to adapt to the way in which developers actually develop software. In other words, most IPSE technologies require fundamental process or activity changes on the part of the user rather than accommodating what the user already does and provide additional functionality in support of their process or activity. Finally, there is concern over long-term support of IPSE approaches.

b. Portable Common Tool Environment (PCTE)

A typical IPSE research initiative is the Portable Common Tool Environment (PCTE), which was a project supported by the European Computer Manufacturers Association (ECMA) during the late 1980’s and early 1990’s. The project

centered on defining a public tool interface for which software engineering environments were constructed. The PCTE provided a specific tool interface and central Object Management System (OMS). Software development tools that complied with this single interface would store their development artifacts within the PCTE OMS. Links (which could be specifically typed) could then be established between objects in the central PCTE database [BOUD88].

In identifying missteps in IPSE research, [BROW92] points out that interface and repository approaches (such as PCTE) are unlikely to be of significant value. The reason for this is because few commercial software development tool developers have substantial economic incentive to build tools compliant with such an interface definition and specific OMS structure.

c. Arcadia

A DARPA IPSE initiative called “Arcadia” was initiated in the late 1980’s with a goal of performing validated research of software development environments. Arcadia consisted of a number of loosely grouped approaches, each of which addressed different parts of the IPSE problem [KADI92a, b]. A few of the systems included in the Arcadia project follow:

- Chiron-1: a user interface system [TAYL94],
- Chimera: a hypertext system for linking heterogeneous systems [ANDE94],
- APPL/A: a software process programming language [SUTT95],
- Pleiades: an IPSE object management system [TARR93], and
- Triton: an IPSE object-oriented database management system [HEIM92].

Arcadia used a process programming language APPL/A to tie together the many threads of an IPSE environment. Much of the work has centered on using and modifying database systems for undertaking syntactic interoperability via the process programming language. Readers interested in the Arcadia project are referred to the collected Arcadia papers in [ARCA95].

d. Weaknesses

The majority of IPSE approaches have focused on supplying syntactic data exchange by providing integration environments and common services for embedded tools. As [BROW92] points out, these efforts have met with little success:

The bottom line is that current work on IPSEs is focusing on the wrong thing. Instead of creating mechanisms for integration at the lexical or syntactic level, it should be addressing how to provide user functionality, productivity, software quality, and so on – issues that come from stressing semantic- and method-level integration.

In some cases, commercial vendors have attempted IPSE implementations by buying collections of CASE tools and “gluing” them together “through a common set of data definitions used between the tools, use of a common data transfer protocol, or through the writing of individual conversion routines to link the tools used by the organization” [BROW93]. Unfortunately, these approaches are not providing substantially new functionality; they are only providing various levels of electronic data interoperability.

The HFSE provides a holistic approach centered on the actual software development process the developer is using. To date, there has been comparatively little research that addresses “process-” oriented IPSE approaches. [BROW93] states:

There is at least one further view of integration which has yet to receive significant attention – a *process* view. This... approach addresses the integration of tools with an organization’s existing software development process. No generally applicable models are currently available in this area.

While Arcadia was a project that attempted to fill a part of this research gap through use of a dedicated, formal process programming language, the HFSE is a less formal (but more flexible and pragmatic) process approach that allows the user to model their existing process graphically and then to holistically link artifacts in CASE tools that the user is already using with a rich set of user defined dependency relationships. Finally, the HFSE differs from this related IPSE work in that it addresses many of the shortcomings of other IPSE approaches because it is relatively light-weight, can be applied incrementally to CASE tools that developers already have (or will purchase in the future), and does not require any long-term support.

4. Software Engineering Ontologies

There is a great deal of literature related to the use of ontologies for capturing the terminology of a domain for software engineering purposes (i.e. to build software tools to support a particular domain). In fact, the Enterprise Ontology presented earlier is one

such example. However, there is very little literature related to the development of ontologies that support the domain of software development tools themselves. Two exceptions worth presenting here are work done for the Software Engineering Body of Knowledge (SWEBOK) and the DARPA Agent Markup Language (DAML).

a. Software Engineering Body of Knowledge

The Software Engineering Body of Knowledge (SWEBOK) is an ongoing IEEE project devoted to providing a "consensually-validated characterization of the bounds of the software engineering discipline" [SWEB01]. The SWEBOK categorizes the existing (and future) knowledge for the domain of software engineering; however, it does not attempt to define that knowledge. The SWEBOK is subdivided into the following ten knowledge areas that discriminate among the important concepts of software engineering:

- Software Requirements,
- Software Design,
- Software Construction,
- Software Testing,
- Software Maintenance,
- Software Configuration Management,
- Software Engineering Management,
- Software Engineering Process,
- Software Engineering Tools and Methods, and
- Software Quality.

Each of these areas is further subdivided by an established taxonomy. For instance, the categories for Software Requirements are shown in Figure 27.

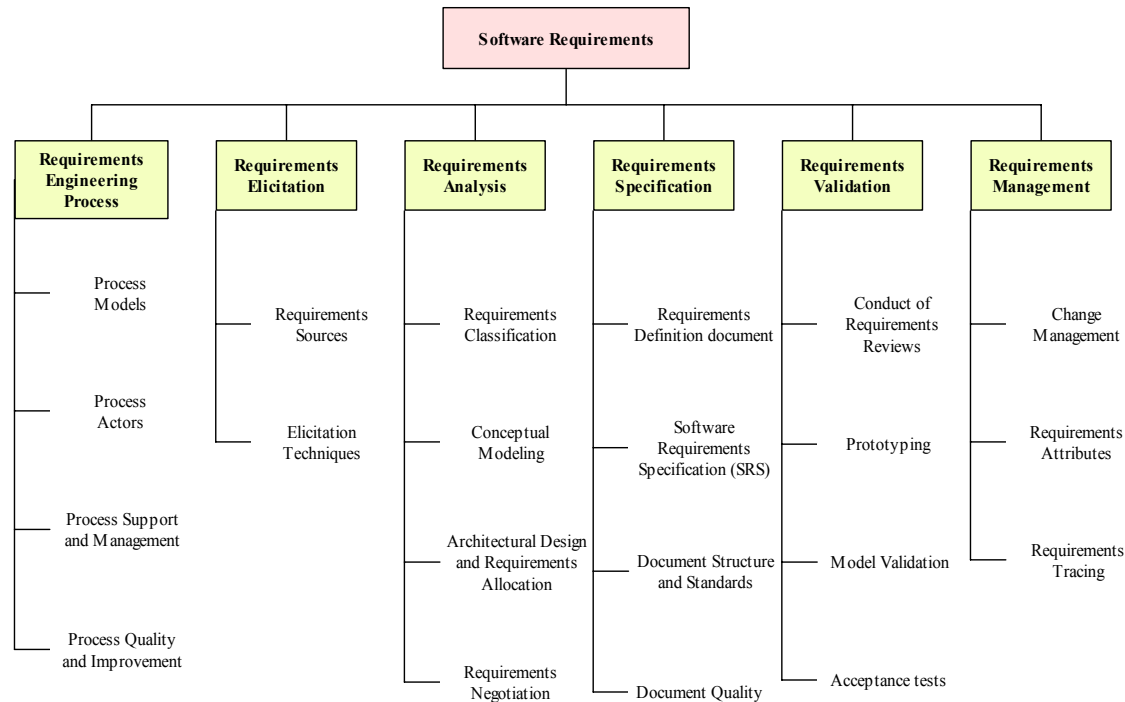


Figure 27 SWEBOK Software Requirements Taxonomy (after [SWEB01])

Note that the level of detail of the taxonomy is fairly abstract. Within the SWEBOK are textual descriptions of what each category of knowledge represents; however, these definitions are not explicit.

More germane to the topic of this dissertation is the SWEBOK taxonomy associated with software development tools. Figure 28 provides the complete listing of the "Tools" portion of the "Software Engineering Tools and Methods" knowledge area.

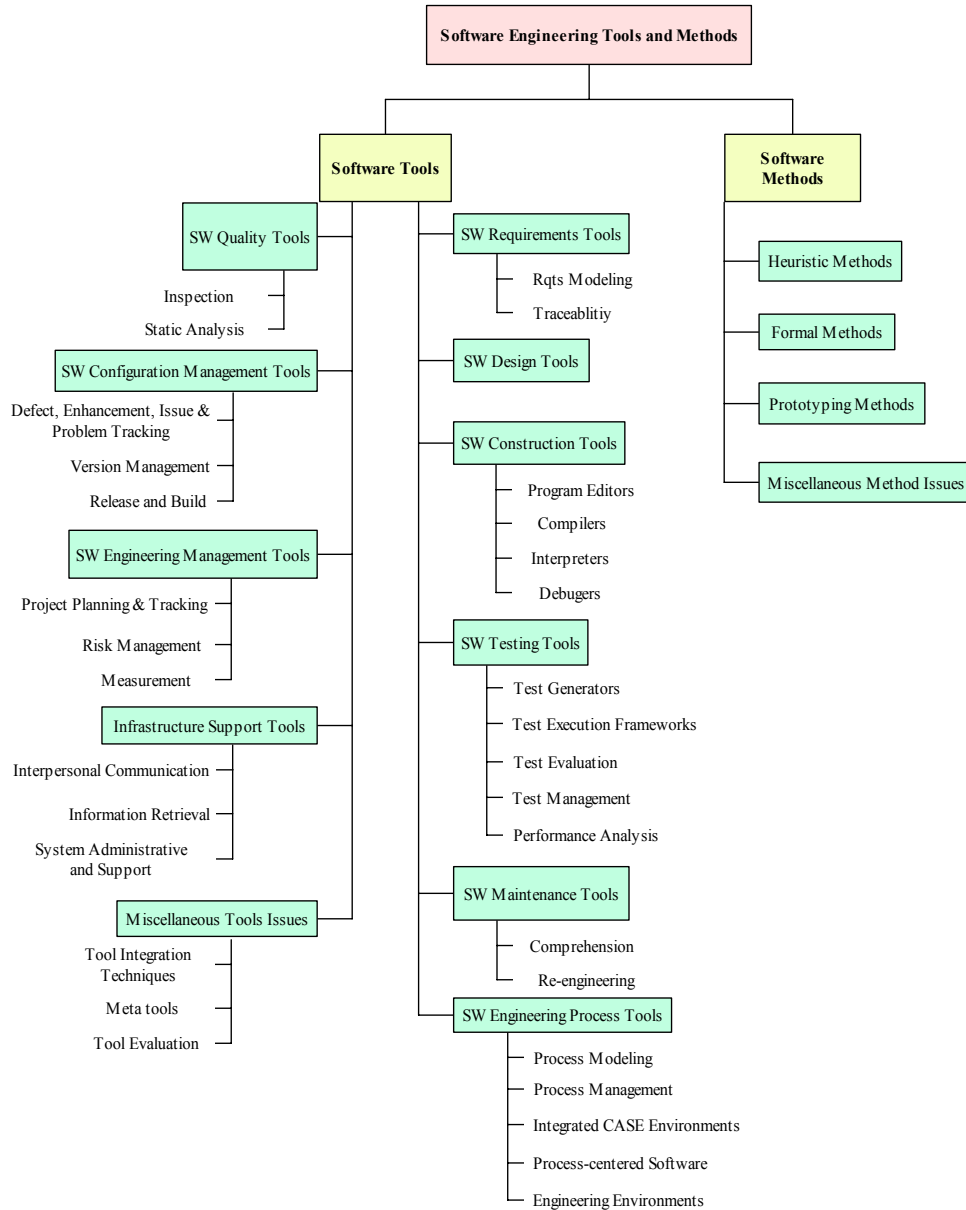


Figure 28 SWEBOK Software Tool Taxonomy (after [SWEB01])

The SWEBOK software tool taxonomy illustrated in Figure 28 can be considered the beginnings of an informal ontology to describe the domain of software development tool knowledge. However, its usefulness as an ontology for establishing interoperability between different software tools is extremely limited. In the case of the HFSE, the SWEBOK does not provide enough detail to make it possible to use this ontology for employing Young's OOMI methodology.

b. *DARPA Agent Markup Language*

The DARPA Agent Markup Language (DAML) project is a relatively recent project that is supporting the development of the “Semantic Web” (an improved World Wide Web where agents can understand the meaning of hyperlinked entities) [DAML03]. One aim of this DARPA program is to link together many ontologies of different domains. In support of this effort, DAML has established an ontology library with over 190 different ontologies from a variety of contributors.

Two of the ontologies in the library deal specifically with software development tools and software engineering. The Software Tool ontology in the DAML library is an ontology developed to provide summary information about the software tools used in a particular research effort. The ontology is relatively small with just four classes and eleven properties. This ontology is summarized in Table 4.

Ontology Purpose	Summary information regarding software tools used by a research program.
Classes in the Ontology	Group, Person, Tool, User
Properties in the Ontology	Category, description, email, howUsed, interface, name, price, site, sourceCode, user, uses

Table 4 DAML Ontology Library: Software Tool Ontology (after [DAML03])

The "Software Engineering" ontology from the DAML library is used to annotate a UML-based toolset. It is a bit larger with over sixty classes and one hundred properties. An excerpt from this ontology is presented in Table 5.

Ontology Purpose	Used to annotate a description of the UML-Based Ontology Toolset project
Classes in the Ontology	AnnotationTool, Annotator, Architecture, ArtificialAgent, ArtificialIntelligence, ArtificialLanguage, CASEtool, CognitiveScience, Component, ComputerScience, DAML, Discipline, domainOntology, Engineering, Feature, feature, FormalLanguage, formalmethods, formalverification, ...
Properties in the Ontology	addsAnnotation, annotates, annotatesWith, appliesSemanticsfrom, applyTo, applyTool, conductedIn, connects, connectsTo, creates, definedBy, designedBy, designedFor, displaysMessage, distribution, elementOf, enablesTool, evaluates, exports, extends, graphicalView, implementationLanguage, implementationOf, implements, imports, includesElements, integrates, interpret, inventedBy, inverseOfaddsAnnotation, inverseOfannotates, inverseOfannotatesWith, inverseOfappliesSemanticsfrom, ...

Table 5 DAML Ontology Library: Software Engineering Ontology (after [DAML03])

Of note is that neither of these existing ontologies really addresses the domain of interest of this dissertation (Software Development Tool Artifacts). The first ontology is only used for collecting summary information about different software development tools that a researcher might use, and the second ontology deals with annotating one specific UML based software development tool. These ontologies provide evidence that previous work has taken place in developing ontologies related to the domain of interest; unfortunately, neither ontology satisfies the need for an ontology to provide software tool interoperability.

5. The Uses of QFD for Software

As previously discussed, the use of QFD in software development has been limited to date. However, there are some case studies worth mentioning. [BETT90] presents a case study from the PRIMA project at Hewlett-Packard in which an abbreviated set of Zultner [ZULT90] matrices are used. That study found that the major value added by the use of the QFD matrices was to aid in planning and decision making portions of the development effort. [SHAR91] presented an overview of results at IBM in which different subsets of Akao's matrices were used and one of the main noted benefits was in getting all segments of the development effort to work in a cohesive manner in meeting customer requirements. [HRON93] reported a case study from Digital's Corporate Telecommunications Software Engineering (CTSE) group that

performed a distributed QFD session between Europe and the U.S. using Video-Teleconferencing. They too, used modified Akao matrices with computer support for completing the tables.

An interesting approach to Object-Oriented Analysis (OOA) using QFD is presented by [LAMI95]. Table 6 below summarizes the matrices recommended for using QFD to identify and quantify the relationships between key artifacts in OOA (an entry in the table indicates the use of a particular matrix).

	Actor Roles	Use Case	Objects	Data Attributes	Classes	Quality Factors
Users	Roles played					
Actor Roles		Tasks performed				
Use Case	Task participants		Entities affected by task	Data needed for task		"Must-be" quality
Demanded Quality		Task priority & user rqts				
Objects			Entity relationships	Class abstractions	Inheritance relationships	

Table 6 QFD Matrices for Performing Object-Oriented Analysis (after [LAMI95])

[LAMI95] cautiously proposes a methodology for using QFD to check to determine whether the IEEE Quality Factors (efficiency, integrity, reliability, survivability, usability, correctness, maintainability, verifiability, expandability, flexibility, interoperability, portability, reusability, etc.) are present in a software design:

Some practitioners of QFD, myself included, are uncomfortable with a standardized set of quality factors. It is risky to rely on a list of factors that are by necessity very general in nature. Designers and engineers will be seduced into believing that they need only consider the "standard quality factors" to do all that is necessary to assure that a quality product has been developed. The history of QFD, if it has shown anything has been that successful designs are those with meticulously incorporate the voice to the customer as it directly relates to the user's experience with the product in actual use. Every attempt of which I am aware to create generic QFD tables, customer demands, or quality characteristics has been a disappointment. The use of generic lists of "-ilities" should be limited to verification and completeness checking of distinct quality table that have

been constructed according [to] accepted QFD practices... Looked at another way, these "-ilities" represent the "must-be" quality (*dissatisfiers* in Kano's terminology.) From this perspective, it is easy to see that this analysis, while useful, is insufficient to assure exciting quality in a system design.

The significant thing to note from these limited case studies is that none have taken an approach similar as to the one accomplished by the HFSE. All of the SQFD studies surveyed have centered on the use of QFD in the requirements or planning phases of the development effort using some subset of already established QFD matrices. None have attempted the approach taken in the HFSE to define QFD matrices based on already existing software development efforts and to integrate these matrices into the software evolution model.

D. CHAPTER SUMMARY

This chapter presented the results of a comprehensive literature review associated with previous work in the dissertation area. The chapter presented foundational work that establishes the underpinnings of the dissertation research. The foundational work of this dissertation rests largely on these following main research areas and researchers:

- Software Evolution [LEHM69, 97, 98],
- Relational Hypergraph Model of Software Evolution [LUQI90] and [HARN99a, 99b, 99c],
- Software Quality Function Deployment [ZULT90, 92, 93],
- Object-Oriented Methodology for Interoperability [YOUN01, 02a, 02b], and
- Use of Ontologies in Interoperability [USCH96, 98].

The chapter also presented an overview of related work in this area. The distinctions between how this related work differs from the work on the HFSE were highlighted.

III. TOWARDS A SOFTWARE DEVELOPMENT TOOL ONTOLOGY

A. CHAPTER OVERVIEW

The first step in applying Young's OOMI methodology to the domain of software development tools is to establish a federation ontology that describes the domain and to establish specific tool ontologies of the tools to be integrated within the FIOM (and thus the HFSE). This effort was specifically undertaken by Hasni [HASN03] in direct support of this dissertation. This chapter summarizes that work. Readers interested in additional detail of the ontologies developed in support of this dissertation are referred to [HASN03].

This chapter provides an overview of the methodology used to develop the software development tool federation ontology. It summarizes the results of the domain analysis undertaken to produce the federation ontology. It presents the details of the federation ontology as well as summarizes the two specific tool ontologies. Finally, the chapter presents results of how the three ontologies inter-relate by using UML to annotate the inter-relationships. It is from these inter-relationships that the OOMI IDE produces translators that can be embedded in middleware to actually exchange data and perform joint task execution.

B. METHODOLOGY FOR BUILDING THE ONTOLOGY

One way to overcome the obstacles posed by lack of interoperability in heterogeneous software development is to establish a unifying contextual framework for the domain. As this contextual framework or “ontology” emerges; people, organizations, and software systems will be able to communicate with more efficiency. [USCH96] points out that engineers must often integrate different ontologies in the same domain to account for legacy systems. To achieve interoperability between systems with different process ontologies, it is first necessary to develop a common ontology applicable to all. This is the approach Young takes in establishing a FIOM in the OOMI

[YOUN02b] and the approach that will be taken for establishing the object model for the HFSE.

Young's object-oriented methodology for establishing interoperability between heterogeneous systems [YOUN02b] allows interaction between the same real-world entities, represented differently in different systems. This approach resolves the differences that exist between different kinds of systems via an establishment of a FIOM. The establishment of such an object federation between existing process models together with the integration of the federation with an extended evolution model, generates inputs and outputs between subordinate software tools and enables them to interoperate (i.e. exchange data and perform joint task execution).

This portion of the dissertation research builds the needed interoperability ontologies by identifying and defining the essential characteristics of two software engineering tools: a requirements engineering tool (Rational Software Corporation's Requisite®Pro, a main-stream, complex, commercial tool) and a software prototyping tool (the Software Engineering Automation Tool suite (SEATools), a research model with tool support for developing executable software prototypes). The approach taken was to construct an initial (but extensible) federation ontology as well as two detailed ontologies related to the specific software process models of the two tools. In designing the federation ontology it was first necessary to analyze the structure, inputs, and outputs of the two individual tools, perform a domain analysis (of this subset of tools) and produce a feature model of that domain. In constructing the specific tool ontologies, the focus was on identifying the classes (and methods) that were needed to pass objects from one tool to another.

Because there is currently no usable interoperability ontology for the domain of software development tools, it was not possible to rely on previous work in designing a federation ontology. Instead, this ontology had to be constructed "from scratch." Fortunately, there were existing methodologies for designing ontologies [USCH96] and [NOYN01]. It was possible to tailor these methodologies to develop a specific methodology for constructing the federation ontology. The tailored ontology development process consists of the following steps: (1) identify the purpose and scope

of the ontology, (2) perform a feature analysis for the domain of software development tools, (3) collect similar characteristics between different feature models, establish affinity relationships, and group commonalities between the two tools to build a federation ontology representing these commonalities and enter this ontology into Protégé, (4) construct the more detailed ontologies for each tool in Protégé, (5) use UML to represent the relationships between the three ontologies, and (6) document the ontologies.

1. Step 1 -- Purpose and Scope of the Ontology

Determining the purpose and scope of the ontology was fairly straightforward given the HFSE research goal and methodology. In this case the purpose for developing software development tool ontologies is to support the federation and component ontologies required in Young's OOMI interoperability methodology. The FIOM created using the ontologies establishes the basic interoperability construct of the HFSE. In terms of scope, the federation ontology must be broad enough to accommodate all potential software development tools, as well as being extensible in case new ontology terms and relationships have to be added later. The specific development tool ontologies must be detailed enough to account for the software processes and objects actually employed by the software development tools. The existing software API (in the case of Requisite®Pro) and the source code classes (in the case of SEATools) to a large extent dictated the level of detail and scope of the tool ontologies.

2. Step 2 -- Feature Modeling

The second step in the ontology design methodology was to perform a domain analysis of software development tools by constructing and then considering the feature models of Requisite®Pro and SEATools.

Feature modeling is a method used to help define software product lines and system families, to identify and manage commonalities and variabilities between products and systems [CZAR00]. Defining a feature model for an existing software tool provides a means to explore, identify, and define the key aspects of the existing software so that these aspects can then be described more fully in an ontology. It is this ontology that can then be used to establish interoperability between the existing software tools.

This approach for the analysis and the investigation of the structure of inputs, outputs, and relationships of a collection of individual software engineering tools can be characterized as a domain analysis (of this subset of tools) and the production of feature model of that domain. Domain engineering focuses on engineering solutions for classes of software systems; it introduces and implements several different kinds of models, such as feature models. The definition of feature models is an important part of the requirements models (developed during the domain analysis). The feature model can be viewed as an abstract representation of functionality found in the domain and thus each feature is a potentially relevant characteristic of the domain -- "potentially" because the feature also has to be considered in light of the purpose and scope of the ontology. Feature models represent an explicit model of a device or system by summarizing the features and the variation points of the device/system. Features in a feature model include the rationale and the stakeholders for each of feature. A feature model for software system captures the reusability and configurability aspects of reusable software. Feature models provide the means to capture the underlying organization of features in a feature diagram. In the case of this dissertation research, the domain analysis and feature models were reverse-engineered from the existing software tools instead of being forward-engineered through examination and consideration of concepts in the domain.

As an example, Figure 29 illustrates a feature model of a how PSDL timing constraints are implemented in SEATools. Such diagrams provide for rich expression of subtle implementations -- note that even though a "Finish Within" and "Minimum Calling Period" are normally required timing constraints, SEATools leaves these as optional and completes them for the user if they are left blank (e.g SEATools calculates $FW=PER$ and $MCP=MRT-MET$ respectively).

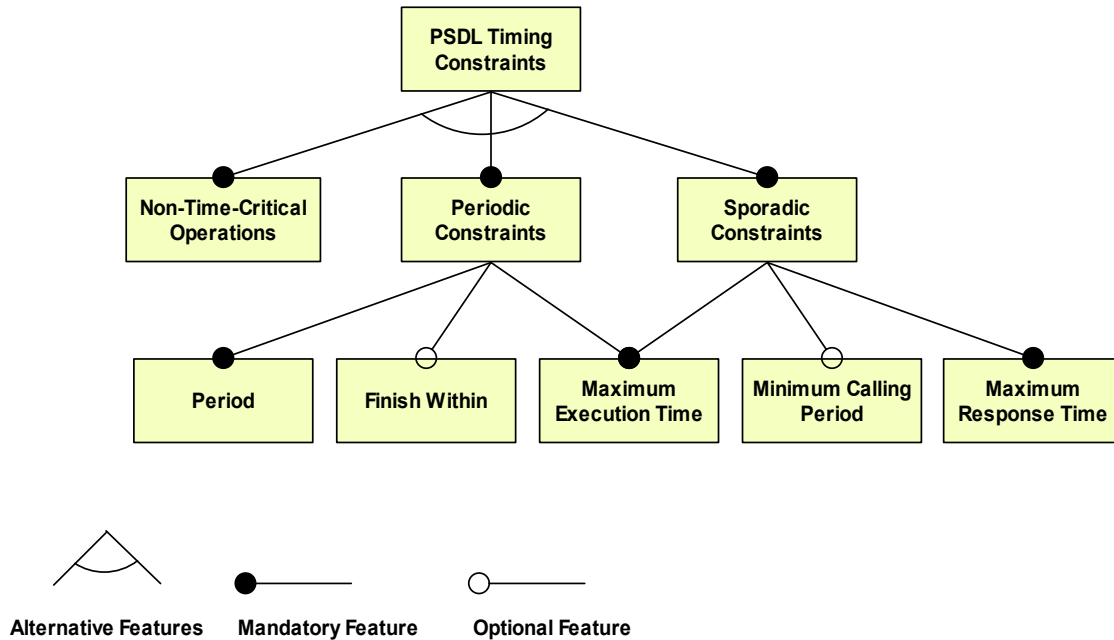


Figure 29 Feature Model of the PSDL Timing Constraints of SEATools (after [HASN03])

The feature model is defined around concepts and not around classes of objects. The objective is to model features of elements and structures of a domain, not just objects in that domain. For more detail on the mechanics of how to construct a Feature model, consult [CZAR00], [GEYE00], or [HASN03].

[CZAR00] provides an excellent methodology for gathering the information needed to construct a feature tree. He identifies the sources of features as the following:

- Existing and potential stakeholders,
- Domain experts and domain literature,
- Existing systems,
- Pre-existing models (e.g., use-case models, object models...), and
- Models created during development (i.e., features gotten during design and implementation).

He goes on to identify the following strategies for identifying and capturing features:

- Look for important domain terminology that implies variability.
- Use feature starter sets to start the analysis.
- Update and maintain feature models during the entire development cycle.
- Identify more features than you initially intend to implement.

[CZAR00] then provides the following set of general steps in feature modeling process:

- Record similarities between instances (i.e. common features).
- Record differences between instances (i.e. variable features).
- Organize the features in feature diagram into hierarchies with classification (mandatory, optional, alternative, and/or optional alternative features).
- Analyze feature combinations and interactions.
- Record all the additional information regarding features.

These steps are referred to as the “micro-cycle” of feature modeling because they are executed in small, repetitive cycles [CZAR00]. While this methodology and strategy was useful in constructing the feature tree for Requisite®Pro and SEATools, this methodology only provided a guide for the actual work. The main difference between this proposed methodology and the actual methodology used centered around the idea that in this particular case, the goal was to "reverse-engineer" feature trees from existing software products; not attempt to define feature trees for prospective software products.

3. Step 3 – Establishing Commonalities

After producing a feature model for Requisite®Pro and SEATools, the next step required was to isolate and annotate the commonalities that exist between the two feature models. These common features then formed the basis for the basic ontology terminology of the software development tool federation. The approach in this step was to reason about the two feature diagrams, develop lists of potential terms from the feature diagrams, identify common terms between the two lists, then construct affinity diagrams of these common terms. Affinity diagrams are hierarchical Venn diagrams that provide groupings of related terms. Figure 30 illustrates how an affinity diagram is constructed; in this case the related terms all deal with "Actors" in the software development process.

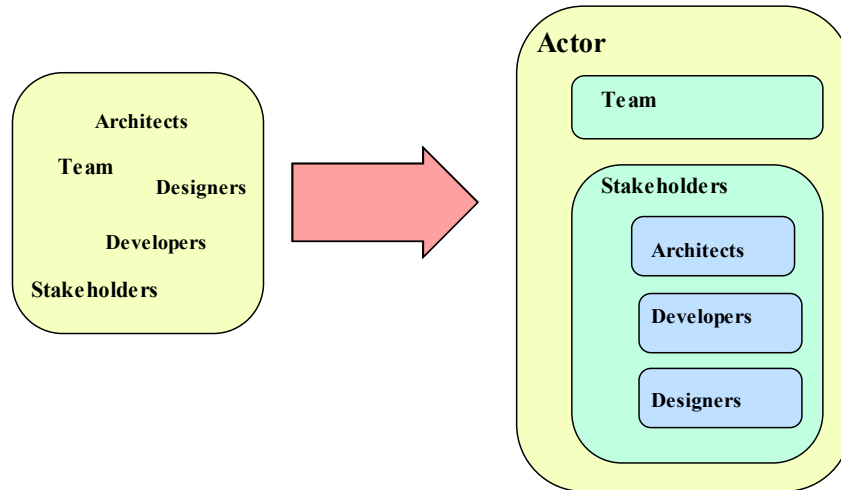


Figure 30 Construction of an Affinity Diagram

The groupings of terms in the affinity diagrams then provided the basis for the hierarchy of terms in the software development tool federation ontology. These terms and hierarchy were then entered and stored in the Protégé-2000 ontology capture tool. This software development tool federation ontology is then left open for further future enhancement and extension.

4. Step 4 – Tool Ontologies

After constructing the federation ontology for the software development tool domain, the next step required construction of the detailed ontologies of the tools to be integrated into the HFSE (Requisite®Pro and SEATools). In the case of the tool ontologies, the detail needed for interoperability was dictated by the detail available through the API or source code (which ever was available) of the tool. For Requisite®Pro (a commercial tool) the tool ontology was derived directly from the Common Object Model (COM) API of the tool (Rational calls this interface the "Requisite®Pro Extensibility Interface"). In the case of SEATools, the source code was available; therefore, the ontology was derived from a selected set of classes and public methods related to the artifacts that were to be transmitted to (or received from) other software tools.

During this ontology construction process, [GRUB95]'s guidelines for ontology construction were adhered to as much as possible: clarity, coherence, extensibility, minimal ontological commitment, and minimal encoding bias. However, because it was

necessary to adhere closely to the actual class constructs of the tools themselves, it was often not possible to satisfy each of these guidelines. In fact, "minimal encoding bias" was not adhered to at all; because to later achieve interoperability, the encoding had to exactly match the class, method, and attribute structures of the API and source code. As the terms of each tool ontology were identified they were input into the Protégé-2000 ontology capture tool. This tool makes it possible to generate XML schemas of the classes in the ontology. These XML schemas are the required input for Young's OOMI IDE.

5. Step 5 - UML Representation of the Domain

The fifth step in the ontology design methodology required that the relationships between all three ontologies be identified and annotated. The reason for this was to formulate inter-relationships between the ontologies so that they conform to the general organization required for Young's OOMI. The basic object structure of OOMI requires that the ontologies between the federation representation of a real world entity and the ontologies of the component representations of the same entities be related through the use of UML. Figure 31 illustrates the general UML structure that was used to annotate how all three ontologies were related.

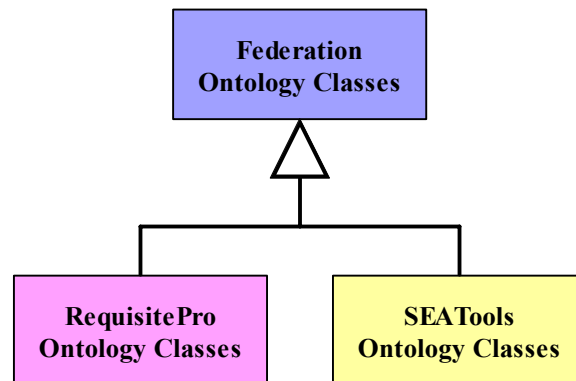


Figure 31 Ontology Inter-relationship (after [HASN03])

Such representations then make it possible to construct a FIOM -- the set of all federation entities in the domain. Based on the FIOM, the OOMI IDE generated translators that were embedded in middleware that actually performed the translations between the two tools.

6. Step 6 -- Documentation

The three ontologies developed for this dissertation research were self-documenting. The purpose, scope, and methodology used for designing the ontologies are presented here in this dissertation and in [HASN03]. The domain analysis and UML diagrams showing the key inter-relationships between the three ontologies are presented in [HASN03]. Excerpts from the complete ontologies are presented in [HASN03] and the complete ontologies themselves are stored in Protégé project files. Such documentation makes it possible for future researchers to modify the ontologies or add additional tools to the ontology set.

As a final note about the methodology used to develop these initial software development tool ontologies, a modified version of this methodology should be used when adding additional tools to the HFSE federation. The modified methodology includes the following:

- In Step 1: Confirm that the purpose is still valid; expand the scope to include the new tool ontologies; remove any existing tool ontologies from the framework that are no longer needed or are invalid.
- In Step 2: Only perform enough feature modeling of the new tools so that needed constructs for the federation ontology are identified. Since the federation ontology is already established it is only necessary to extend and modify it, not re-build it entirely.
- In Step 3: Modify the federation ontology to account for the new/modified ontology terms from Step 2.
- In Step 4: Perform the same Step 4 methodology as delineated above.
- In Step 5: Modify each UML relationship diagram as needed to account for the new tool ontologies and the changes to the federation ontology identified in Step 3.
- In Step 6: Perform the same Step 6 methodology as delineated above.

This modified methodology makes it possible for future researchers to easily add additional software development tools to the HFSE.

C. DOMAIN ANALYSIS AND FEATURE MODELS

In Step 2, of the methodology above, a domain analysis was undertaken of the software development tool domain. The analysis was accomplished by examining two

specific software development tools, building feature models of those tools, and then identifying key terminology of the feature models. There are two reasons why this domain analysis cannot be considered to be a complete analysis of the domain of software development tools. First, only two tools (out of many hundreds of possibilities) were analyzed. Secondly, a domain analysis is not an “additive” activity; simply analyzing additional tools (beyond those two) by themselves does not completely add to the overall analysis. The ways in which the new additions affect and change the previously established analysis must also be considered. Therefore, the limited domain analysis conducted as part of this research can be considered a necessary, but not sufficient, analysis towards establishing the HFSE.

1. Rational Requisite®Pro

The first tool analyzed in the domain analysis was Rational Corporation's Requisite®Pro, a large commercial requirements management tool. The Feature Model was developed by identifying software features from the Requisite®Pro User's Guide [RATI01] and by actual day-to-day use of the tool. See [HASN03] for the complete Feature Model for Requisite®Pro. Figure 32 illustrates a single excerpt from the overall Feature Model for Requisite®Pro.

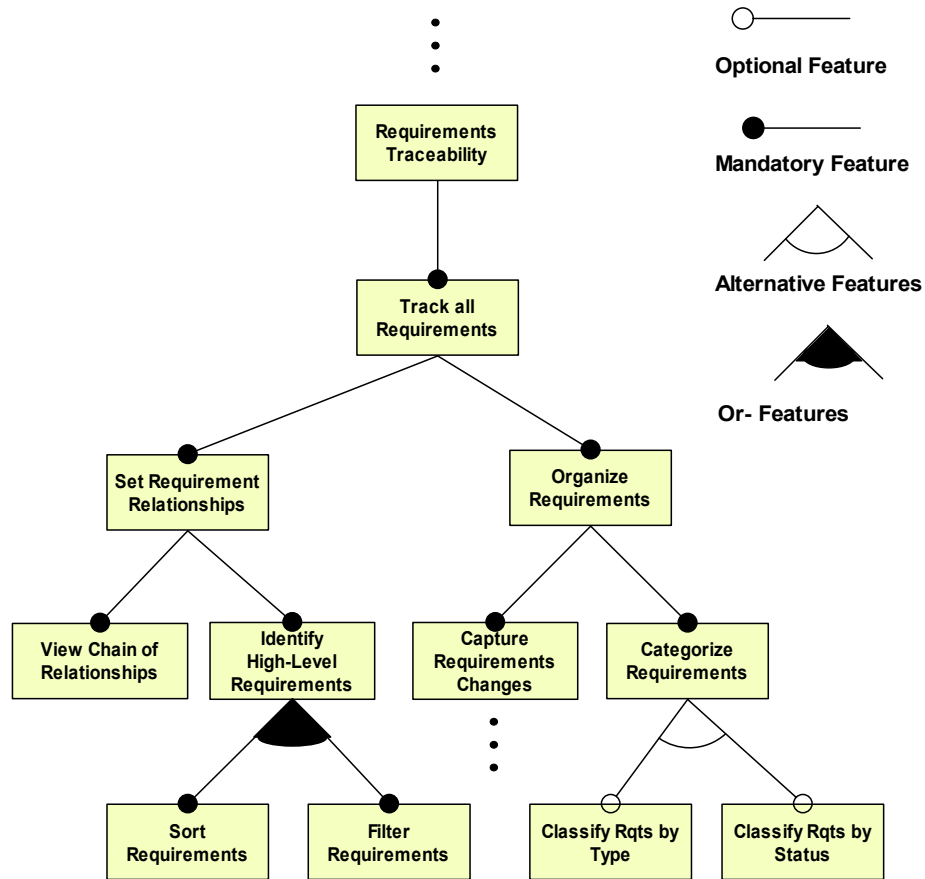


Figure 32 Excerpt of the Requisite®Pro Feature Model (after [HASN03])

This excerpt illustrates the features associated with the requirements' traceability functionality of a Requisite®Pro and is just a small portion of the total functionality (a small subset of the features) of the entire tool.

From the complete Feature Model of Requisite®Pro, it was then possible to extract relevant features (with their descriptions). Each of these features then becomes a candidate for possible inclusion in the federation ontology. The complete list of Requisite®Pro features is listed below in Table 7 [HASN03].

Ref #	Feature	Description
1	Requisite®Pro	A requirements management tool
2	Management	Documenting and managing requirements throughout the development lifecycle
3	Requirements analysis	Requirements linking, tracing, and report generation

Ref #	Feature	Description
4	Non-functional features	The subset of non-functional features of the tool (e.g. integration with other tools, security, and remote usage via web)
5	Manage projects	Projects are the top-level objects managed by Requisite®Pro
6	Manage teams	Group members of the project team for working in a collaborative environment
7	Manage documents	Capture, communicate, organize, and track document information
8	Set up new project template	Creates new project templates from existing projects
9	Remove a project from project list	Remove projects from project list
10	Allow project revision	Allow the revision of the project
11	Unify teams	Unify project managers, QA managers, testers, developers, etc. in communicating and managing systems requirements
12	Allow Interaction with stakeholders	Records stakeholder communications and decisions made about requirements
13	Provide standard project templates	Customers can use Rational Requisite®Pro's predefined project structures or define their own
14	Report statistics	Requirement metrics provide project managers with statistics to be displayed in Excel
15	Provide isolated database	Each project is maintained in its own sub-directory
16	Synchronize textual Software Requirements Specification (SRS)	Synchronize textual SRS with database contents
17	Manual revision of the project	Allow manual revision of the project
18	Automatic revision of the project	Allow automatic revision of the project
19	Notify teams	Keep everyone informed of the current requirements information
20	Discuss and query	Enables threaded discussions on requirements
21	Provide collaborative design environment	Allows collaborative discussions among the team
22	Record comments	Provides recording mechanisms, saves communications to the project database
23	Provide Consistency	Consistency is checked by other members of the collaborative team

Ref #	Feature	Description
24	Provide Synchronization	Requirements database is continually updated as new information is entered and recorded
25	Improve Efficiency	Provides mechanisms for better communication
26	Improve Understandability	Team members are informed of the current requirements information with traceability to early design decisions
27	Improve Effectiveness	Optimizes team collaboration around the requirements
28	Easy Access to documents	Provide access to all requirements for every team member, by using a central database
29	Customize user documentation	Customers tailor documentation to their roles and preferences
30	Maintain documents	Provides a document repository
31	Archive	Allows the archiving of old documentation
32	Detect documentation changes	Automatically detects changes to existing documentation
33	Monitor links	Defines traceability relationships or links between individual requirements and between requirements and other system elements
34	Set up links	Create relationships between artifacts in either the Word or View Workplaces
35	Identify and clear suspect links	Relationships between previously linked requirements are marked as suspect if the text, type, or attributes of either requirement is changed. This relationship can be cleared in either Word or View workplaces
36	Automatic set to “suspect”	Allows links to be automatically set to “suspect”
37	Manual set to “suspect”	Allows links to be manually set to “suspect”
38	Automatically clear suspect links	Provides automatic clearing of suspect links
39	Manually clear suspect links	Provides manual clearing of suspect links
40	Provide traceability	Provides views of chained relationships between requirements
41	Control requirements	Controls the access of multiple users, which provides control at both the project and document level
42	Create requirements	Creates requirements through Word or a View Workplace
43	Edit requirements	Edits requirements through Word or a View Workplace
44	Verify requirements	Ensures that requirements serve as direct input to test creation

Ref #	Feature	Description
45	Update requirements	Updates the Word Workplace when the requirement text in the document is modified and the document is saved
46	Add requirements	Adds requirements to the requirements database
47	Delete requirements	Deletes requirements or requirement attributes without disrupting work elsewhere
48	Provide requirements' type	Defines different types of requirements
49	Assign attributes to requirements	Defines different attributes for different types of requirements and set attribute values for individual requirements
50	Prioritize requirements	Provides a prioritization attribute
51	Relocate previous requirements	Relocates previous requirements
52	Save requirements	Saves requirements to the project database
53	Label Requirements temporarily	Provides a "change pending" function, until the change is appropriately approved
54	Uniquely identify requirements	Assigns a unique identifier to each requirement
55	Facilitates requirements coverage analysis	Developers can assess whether they have documented in detail all features
56	View approved use-case	Connects requirements with use-case models instantly accessible by developers. It help to ensure that the implemented functionality reflects the customer needs
57	Track all requirements	Provides views that track the status and attributes of all requirements
58	Set requirements relationships	Establish relationships among requirements
59	Organize Requirements	Organizes requirements by type
60	Establish requirement hierarchies	Arranges the requirements' attributes in a hierarchy
61	View chain of relationships	Views requirements' chain of relationships
62	Sort the requirements	Sorts requirements according to user specified attributes
63	Filter the requirements	Filters requirements according to user specified attributes
64	Facilitate the Understanding of the impact of changes	Provides views for impact analysis tailored to each team member

Ref #	Feature	Description
65	Report generation	Automatically generates user defined reports
66	Tailors usability options	Provides the ability to specific and set usability options
67	Remote use via web	Includes web interface for database query, discussion, and for updates to requirement attributes
68	Provides tutorial	Includes learning aids, such as tutorial and/or sample projects
69	Word environment and import wizard	Allows extraction of textual requirements from external Word documents
70	Integration with software tools	Integrates with other Rational tools, such as testing, design, and project management
71	Reduce errors	Collaborative environment helps ensure that errors are identified early and fully corrected
72	Provides Security mechanisms	Permissions to access particular features are assigned to specific groups
73	Finds current version of document	Web access provides stakeholders the most-up-to-date requirements
74	Facilitates contextual understanding	Allows the user to capture information about the context from which a requirement has been derived
75	Set user security privileges	Defines users and groups and their access privileges
76	Lock documents	Applies locking to selected documents

Table 7 Requisite®Pro Feature List (after [HASN03])

This feature list is taken directly from the Requisite®Pro Feature Model. The features towards the beginning of the list are high-level "parent" features, while those towards the bottom represent more detailed "atomic" features.

2. SEATools

The second tool analyzed during the domain analysis was the Naval Postgraduate School's Software Engineering Automation Center's (SEAC) Software Engineering Automation Tool Suite (SEATools). This suite is a research oriented set of prototyping tools for designing and building executable software prototypes of large complex, real-time software systems. The SEATools Feature Model was developed by identifying software features from the SEATools descriptions [LUQI88, 91a, 91b, 96] and by actual day-to-day use of the suite. See [HASN03] for the complete Feature Model for SEATools. Figure 33 below and the earlier presented Figure 29 illustrate excerpts from the overall Feature Model for SEATools.

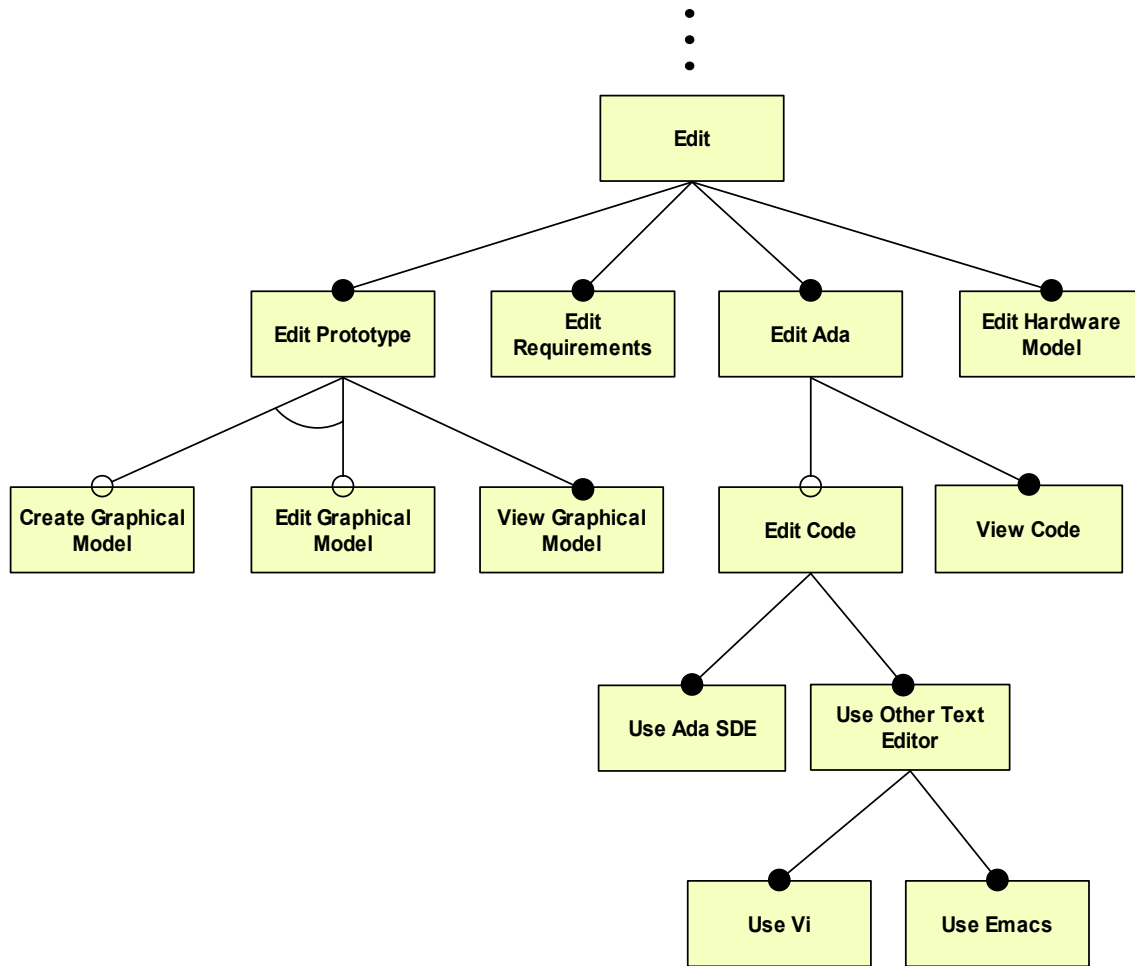


Figure 33 Excerpt from the SEATools Feature Model (after [HASN03])

This excerpt illustrates the features associated with the edit functionality of a SEATools and is just a small portion of the total functionality (a small subset of the features) of the entire suite.

From the complete Feature Model SEATools, it was then possible to extract relevant features (with their descriptions). Each of these features then becomes a candidate for possible inclusion in the federation ontology. The complete list of SEATools features is listed below in Table 8 [HASN03].

Ref #	Feature	Description
1	SEATools	An integrated set of software engineering tools for developing prototypes of real-time systems
2	Management prototype	Manage prototypes developed in SEATools
3	Build prototype	Constructs the prototype (model, code, etc.)

Ref #	Feature	Description
4	User interface	Provides an interface to accept user commands and provide information to the user
5	Develop systems	Develop functional prototypes
6	Analyze requirements	Analyze requirements through evolutionary prototypes
7	Generate code	Automatic generation of the code
8	Model editor	SEATools editor that provides a user the interface to create a software model
9	Modification	Modify existing prototypes and graphs
10	Graphical editor	Edits the graphical view of the software model
11	Expert-system design mode	Provides a user interface that allows the user to access SEATools
12	Debugger	Identifies bugs in the software model
13	Browser	Provides navigability to different portions of the software model
14	Evolutionary prototype	Maintains evolutionary prototype versions and variants
15	Feasibility	Supports software feasibility studies through prototype construction
16	Project control	Assures control of projects via the use of merger
17	Interaction	Allows interaction with the proposed system with its environment
18	Constraints	Allows users to input timing constraints
19	Software base	One of the five categories of the SEATools software
20	Execution support system	The window in which SEATools is initially invoked
21	Creation	Allows the creation of a prototype, PSDL, and graphs.
22	Add	Allows the adding of information to an existing prototype
23	Refine systems	Allows changes in an existing prototype
24	Deletion	Allows the deletion of undesired information
25	Allow communication	Allows communication between different parts of the model
26	Control communication	Controls communication between different parts in the model
27	Tools	Differentiates tools
28	Integration of complex systems	Supports integration of complex systems
29	Design	Assessment of design
30	Evolution control systems	Provides automated support for coordinating the multiple versions of design

Ref #	Feature	Description
31	Merger	Provides automated prototype change-merging
32	Subsystems	Allows users to generate subsystems
33	Software design	Manages the software design
34	Design base	Provides persistent storage of the prototype development data
35	Translator	Translates PSDL into Ada code
36	Scheduler	Creates schedules for timing requirements
37	Compiler	Compiles the source code
38	Execute system	Executes all the Ada code for the currently open prototype
39	Designer	Designs a prototype
40	User	One of the potential stakeholders in a project
41	Prototype	A software model that implements some subset of requirements for later delivered software system
42	Help	Assist the user/software engineer when requesting information about one of the menu buttons
43	Edit	Provides the ability to edit portions of the prototype (PSDL, Ada,, Requirements...)
44	Essential	A category of differentiation for user interfaces, editors, the execution support system, the project control system, and the software base
45	Very useful	A category of differentiation for user interfaces, editors, the execution support system, the project control system, and the software base
46	Useful	A category of differentiation for user interfaces, editors, the execution support system, the project control system, and the software base
47	Conflict detection	Detect timing conflicts
48	Warning	Warns of any existing conflict
49	Design database containing PSDL	Contains the PSDL descriptions and working code for all available reusable software components
50	Construction	Allows the construction of a prototype
51	New	Allows the user to create a new prototype
52	Quitting	Quits and closes the SEATools program
53	Commit work	Allows prototype design to be entered into the database
54	Retrieve from database	Allows the user to retrieve data from the database
55	Choice	Allow the choice of project type
56	PSDL	Construct prototypes using a combination of graphical and textual objects
57	Interface	Invokes Transportable Applications Environment Plus (TAE+) to edit the prototype interface

Ref #	Feature	Description
58	Requirements	Allows designers to edit a requirements file
59	Ada	Allows designers to edit Ada implementation files
60	Caps default	Allows designers to choose which text or Ada editor will be used
61	Hardware model	Lets designers check timing constraints relative to a machine faster or slower than the machine that is executing CAPS
62	Operating systems	Allows designers to account for operating system in the prototype design
63	Assembler	Allows designers to account for operating system the assembler in the prototype design
64	Programming language	Allows designers to account for programming language in the prototype design
65	Computer systems	Allows designers to account for hardware in the prototype design
66	Libraries	Provides libraries for use in prototypes
67	Editors	Provides prototype editors
68	PSDL specifications	Track PSDL specifications
69	Executed code	Track executed code
70	Graphical objects (data flow diagram)	Allow the construction of data flow diagram
71	Textual objects	Constructs and edits textual objects
72	Data flow diagram	Shows existing data flow diagram
73	Computational graphs	Constructs and edits computational graphs
74	Finding	finds prototype graphs
75	Retrieval	Retrieve prototype from the database
76	Graphical design	Create graphical design
77	Edit graphical design	Edit graphical design
78	View graphical design	View graphical design
79	View code	View code
80	Edit code	Edit code
81	Library reused code	Use the Reuse Library
82	Control constraints	Controls the process and output generation via a set of conditions or predicates
83	Operators	Allows the drawing of operators (circles) in a data flow diagram
84	Streams	Allows the drawing of data streams (directed lines) in a data flow diagram
85	Terminator	Allows the drawing of terminators (rectangles) in a data flow diagram
86	Timing constraints	Allows the entry of Timing constraints
87	Ada SDE	A text editor for editing Ada code

Ref #	Feature	Description
88	Other text editor	Used to view and edit text and code
89	Vi	A text editor for editing Ada code
90	Emacs	A text editor for editing Ada code

Table 8 SEATools Feature List (after [HASN03])

This feature list is taken directly from the SEATools Feature Model. As in the case of the Requisite®Pro feature list, the features towards the beginning of the list are high-level "parent" features, while those towards the bottom represent more detailed "atomic" features.

D. FEDERATION ONTOLOGY

After performing a domain analysis using an in-depth investigation of Requisite®Pro and SEATools, the two lists were considered together to identify commonalities -- commonalities that would also likely be common with other software development tools. These commonalities begin to form the list of terms that eventually will make up the federation ontology. Table 9 lists the common terms from the domain analysis [HAS03].

Ref #	Feature	Description
1	Tool	A software development tool
2	Actor	Individual(s) participating in one or more roles in a software development effort
3	Stakeholders	A person, group, or organization with a stake in the outcome of an application that is being developed
4	Developers	The software engineers who develop a software system
5	Designers	The software engineers who design a software system
6	Architects	The software architects for a particular software system
7	Team	A team involved in any software project
8	Activity	A sequence of actions undertaken by actors or the tool
9	Communication	Transmission and receipt of information
10	Management	Control and direction over all or part of a software development effort
11	Organization	Arrangement of the software artifacts or information related to a software project

Ref #	Feature	Description
12	Sorting	Arrangement of the software artifacts or information related to a software project using user defined criteria
13	Filtering	Removal of undesired information using user defined criteria
14	Synchronization	Provides software project stakeholders and information to operate at the same rate and time
15	Archiving	Storage of particular information related to software activities
16	Maintenance	The process of repair, modification, and enhancement of a system
17	Creation	The initial formation of a software artifact
18	Coding	The activity actors perform to generate Code
19	Modification	Changes to an existing artifact
20	Verification	Confirming that an artifact is correct
21	Artifact	Data, source code, or information produced, gathered or used during the software development process
22	Documentation	Recorded information about the software development process
23	Statistics	Numerical data related to the software development process
24	Database	A collection of arranged data available for easy and fast retrieval
25	Feedback	The provision of information to actors for comparison purposes
26	Efficiency	Improved activity
27	Links/Dependencies/ Traceability	Relationships between the different artifacts in a software development effort
28	Security	A type of dependency focused on the avoidance of risk and danger
29	Child Parent	A type of dependency focused on a hierarchical arrangement of artifacts
30	Risk	The chance of damage or loss
31	Safety	Freedom from damage or loss
32	Project Component	An individual entity within a
33	Requirements	A statement of what functionality, appearance, and behavior are required of a software system
34	Model	A view of the design of an application from a particular perspective

Ref #	Feature	Description
35	Use Case	A model of an actors interaction with a software system
36	Library	A collection of information and material related to a project
37	Prototype	A partial implementation of a software system implemented for a particular purpose (e.g. confirm requirements, test feasibility of technology, etc).
38	Test	Assure the determination, the quality, and the truth of a software system

Table 9 Common Characteristics for Software Development Tool Federation (after [HASN03])

After identifying these common terms in the domain of software development tools, the terms were organized into logical groupings using an "affinity diagram" technique (recall Figure 30). From these affinity diagrams, it was then straight-forward to establish the hierarchical structure of the federation ontology. The completed federation ontology is shown below in Figure 34.

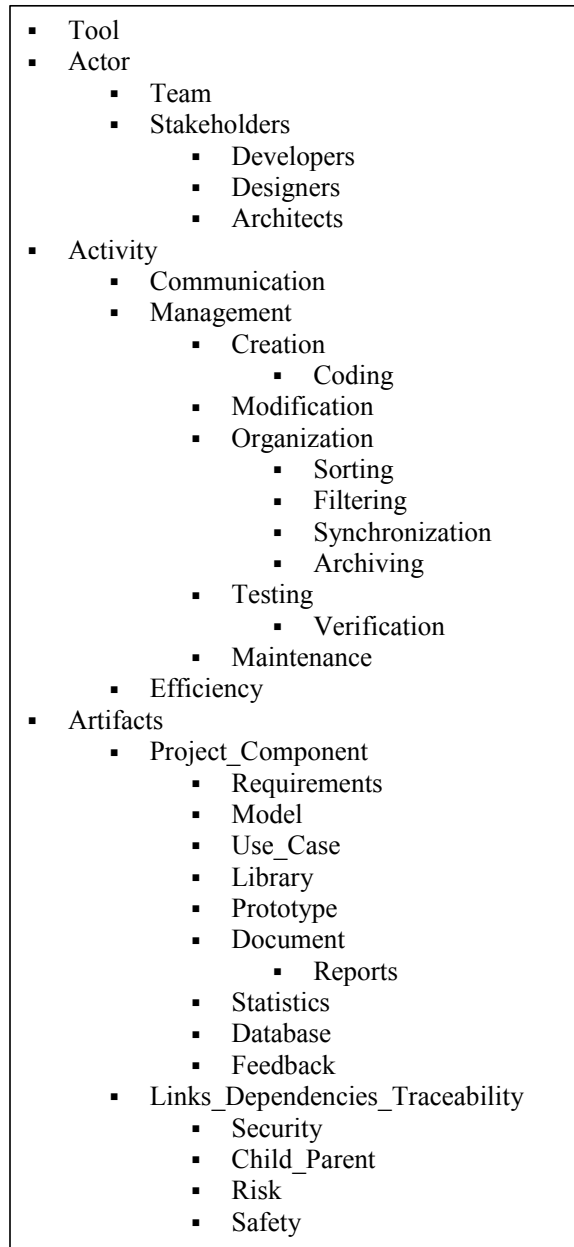


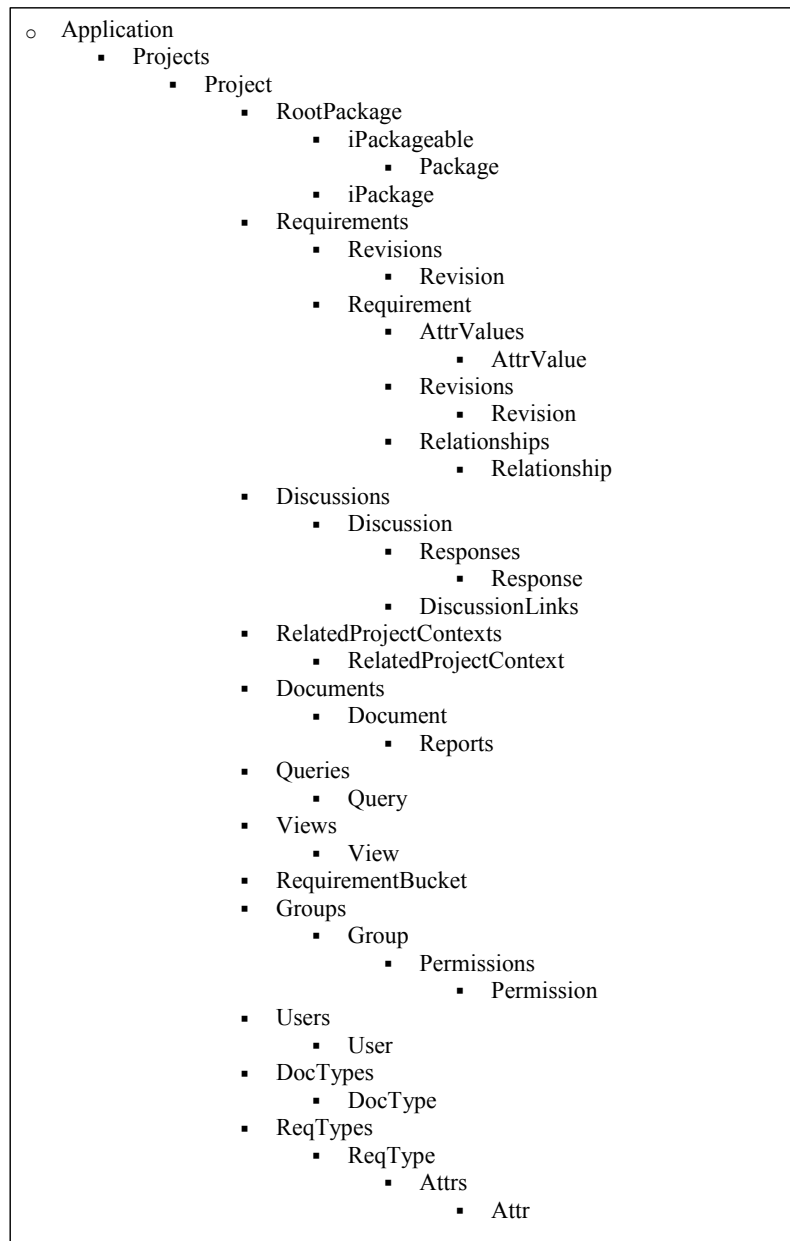
Figure 34 Software Development Tool Federation Ontology (after [HASN03])

The final activity after formulating this hierarchy was to input the ontology into Protégé. From protégé it was then possible to generate XML schemas that served as the input to the OOMI IDE.

E. TOOL ONTOLOGIES

After developing the federation ontology, the next step was to develop the individual tool ontologies of the tools that were to be integrated into the HFSE. The level

of detail of these ontologies was dictated by the detail of the classes to be used to establish interoperability between the tools. In the case of Requisite®Pro this came from Requisite®Pro's Extensibility Interface (its COM API). Figure 35 illustrates the structure of about 75% of the Requisite®Pro ontology. Each of these classes has numerous attributes and methods (e.g. the Requirement Class has sixty attributes and methods). See [HASN03] for the complete ontology.



**Figure 35 Excerpt of the Class Structure of the Requisite®Pro Ontology
(after [HASN03])**

In the case of SEATools, the source code was available so the classes came directly from a subset of the source code. This subset was identified by reverse-engineering the source code in TogetherSoft's (acquired by Borland in January 2003) Together 6.0 Java IDE [TOGE03]. Within Together, a subset of classes and public methods was identified for the objects that were intended to be passed to other software applications. This subset formed the basis of the SEATools ontology. Figure 36 illustrates the complete structure of the SEATools ontology. Each of these classes has several attributes, methods, and properties. See [HASN03] for the complete ontology listing.

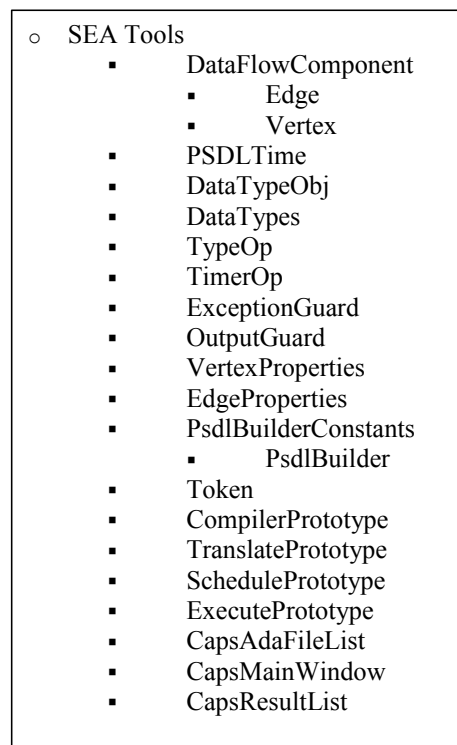


Figure 36 Class Structure of the SEATools Ontology (after [HASN03])

Together, these two tool ontologies form the basis for the component representations within Young's OOMI [YOUN02b]. The OOMI FIOM was constructed by establishing the relationships between these two ontologies and the software development tool federation ontology through the use of UML.

F. ONTOLOGY INTER-RELATIONSHIPS

After the federation ontology and the two component ontologies (tool ontologies) were defined, the next step in the methodology (step 5) required that the relationships between the three ontologies be identified and annotated. This was done using UML. Both a top down and bottom up approach were taken to identify the relationships between the three ontologies and record those relationships in static class diagrams (recall Figure 31). In Young's OOMI methodology, the interoperability engineer and the ontology manager determine the number of real-world entities to relate based on the types of interoperability to be achieved. In support of this dissertation, eight such relationships were established [HASN03]. Figure 37 is an example of how the three ontologies are related for the relationship of "Communication."

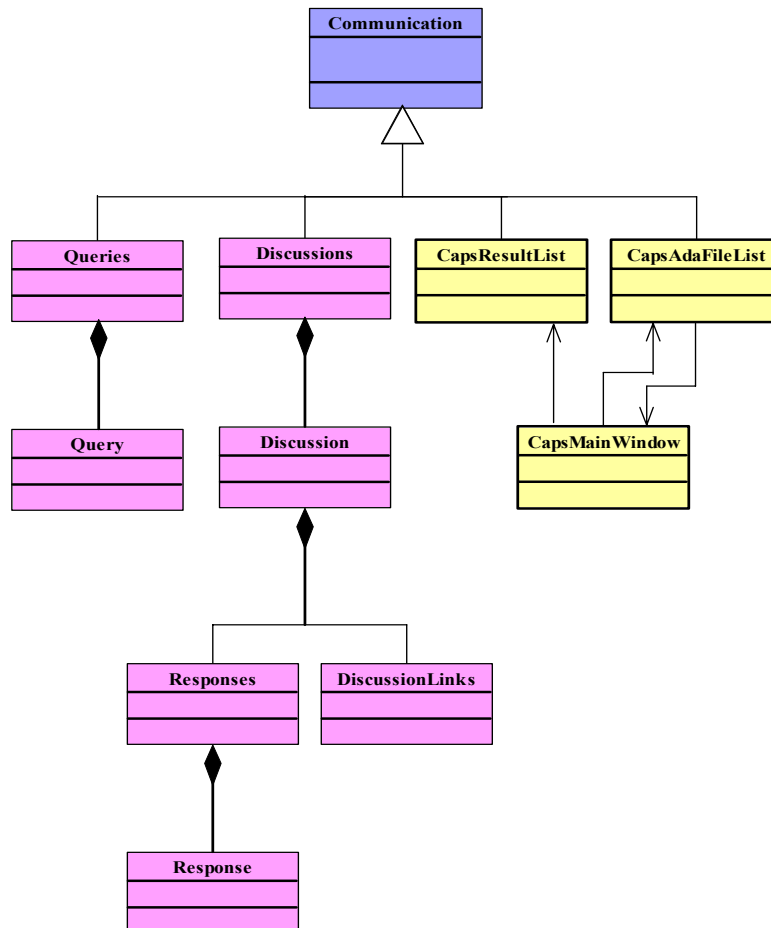


Figure 37 Communication Class Inter-relationships (from [HASN03])

Figure 37 is just one example of how the relationships between the three ontologies are related. See [HASN03] for the remaining sets of relationships.

G. CHAPTER SUMMARY

This chapter presented the methodology and results of the research effort devoted to establishing the set of software development tool ontologies for integration into the HFSE. A six-step methodology was defined and used to develop a federation ontology and two specific tool ontologies. Together these three ontologies form the basis for the establishment of a FIOM using Young's OOMI methodology [YOUN02b].

The six step methodology is extensible so that additional tools can be later integrated into the framework by future researchers.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. INTEGRATING QUALITY FUNCTION DEPLOYMENT INTO THE RELATIONAL HYPERGRAPH MODEL OF SOFTWARE EVOLUTION

A. RELATIONAL HYPERGRAPH SOFTWARE EVOLUTION MODEL

1. Overview of the Relational Hypergraph Software Evolution Model

As previously discussed, Harn establishes a *Relational Hypergraph model (RH model)* to describe Software Evolution. This model establishes dependencies and links between key activities/artifacts of a software development cycle and also between sequential iterations and variations of cycles. The model allows the development of tools to manage both the activities in a software development project and the products that those activities produce.

In the Relational Hypergraph model, activities and artifacts affected by the software evolution process are called software evolution objects and consist of "Steps" and "Components." The Relational Hypergraph uses a hierarchical refinement (Top-level objects, refined objects, atomic objects) to link these objects and establish dependencies (both primary dependencies and secondary dependencies) between the objects.

Dependency Rules are recorded within the object attributes. For instance, step attributes might consist of the following: version and variation number, status, predecessor, priority, deadline, estimated duration, earliest start time, finish time, evaluation, manager, organizer, evaluator. Component attributes might consist of the following: version and variation number, hypertext, code, data, pictures, charts, movies, etc.

2. Important Definitions in the RH Model

Harn provides the following definitions of key entities within the RH model of software evolution [HARN99b]. Since the RH model forms the core of how software development artifacts are represented within the HFSE, the following definitions provide a foundation for the entities used in the HFSE:

Definition 1. (Hypergraph) [HARN99b] applied the [BERG89] definition of the hypergraph as follows: A *(directed) hypergraph* is a tuple $H = (N, E, I, O)$ where

1. N is a set of *nodes*,
2. E is a set of *hyperedges*,
3. $I : E \rightarrow 2^N$ is a function giving the set of *inputs* of each hyperedge, and
4. $O : E \rightarrow 2^N$ is a function giving the set of *outputs* of each hyperedge.

Definition 2. (Evolutionary Hypergraph) [HARN99b] An *evolutionary hypergraph* is a labeled, directed, and acyclic hypergraph $H = (N, E, I, O)$ together with labeling functions $L_N : N \rightarrow C$ and $L_E : E \rightarrow A$ such that the following is true:

1. The elements of N represent unique identifiers for software evolution components,
2. The elements of E represent unique identifiers for software evolution steps,
3. The functions I and O give the inputs and outputs of each software evolution step, such that $O(e) \cap O(e') \neq \emptyset \Rightarrow e = e'$,
4. The function L_N labels each node with component attributes from the set C , including the corresponding version of the software evolution component, and
5. The function L_E labels each edge with step attributes from the set A , including the current status of the software evolution step, such that $A = \{s, d\} \cdot A'$ (that is, each element of A has the form (s, a') or (d, a') , where $a' \in A'$).

Definition 3. (Relational Hypergraph) [HARN99b] An evolutionary hypergraph $H = (N, E, I, O)$ is called a *relational hypergraph* if and only if for every hyperedge e in H and every input node n in $I(e)$, the relationship between n and e is *primary_input* or *secondary_input*.

Definition 4. (Primary and Secondary Dependency) [HARN99c] If an input node and an output node of an evolutionary hyperedge are different versions of the same component, then the path from the input node via the hyperedge to the output node of the step is called a *primary-input-driven path*, and the relationship between the input node and the step is called a *primary_input* dependency. If an input node and an output node of an evolutionary hyperedge are different components, then the path from the input node

via the hyperedge to the output node is called a *secondary-input driven path*, and the relationship between the input node and the step is called a *secondary_input dependency*.

As an example, Figure 38 illustrates a portion of a Relational Hypergraph model for a software system.

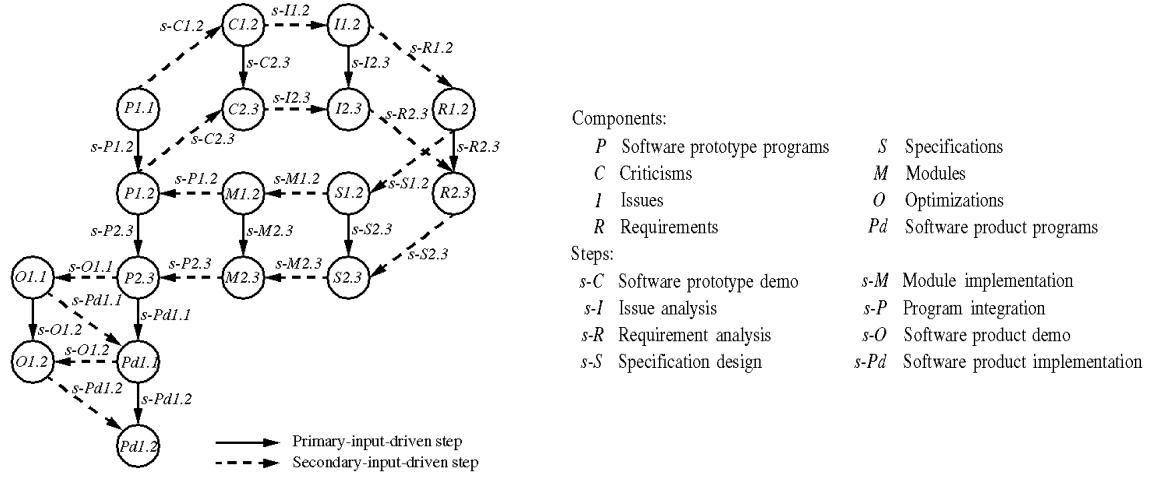


Figure 38 Sample Relational Hypergraph (from [HARN99c])

Definition 5. (Top-Level Evolution Step) [HARN99c] Let $H = (N, E, I, O)$ be an evolutionary hypergraph. A hyperedge $e \in E$ is called a *top-level evolution step* if and only if the hyperedge e has no parent evolution step.

Definition 6. (Atomic Evolution Step) [HARN99c] Let $H = (N, E, I, O)$ be an evolutionary hypergraph. A hyperedge $e \in E$ is called an *atomic evolution step* if and only if the hyperedge e cannot be expanded to additional steps and its output set has at most one component.

Definition 7. (Top-level Evolutionary Hypergraph) [HARN99c] A *top-level evolutionary hypergraph* is an evolutionary hypergraph $H = (N, E, I, O)$, each of whose hyperedges is a top-level evolution step.

3. Embedding QFD within the Relational Hypergraph Software Evolution Model

Actually embedding key portions of the QFD methodology within the Relational Hypergraph model requires several additions and changes to the RH model of software evolution proposed by [HARN99c].

a. *Project Schema*

The HFSE relies on a project schema as an initial basis for a particular software development process. This schema can be viewed as an abstract representation of the different types of artifacts and different activities within a software development effort. The “Waterfall Model,” “Spiral Model,” and “Evolutionary Process Model” are general examples of project schemas; however, a software engineer should explicitly identify the types of artifacts (components) and activities (steps) within their particular HFSE project schema. Formally, a project schema is a top-level evolutionary hypergraph expressed as follows:

Definition 8. (Project Schema) A *project schema* is the top-level evolutionary hypergraph $H = (N, E, I, O)$ of a particular software development effort.

b. *QFD Dependency*

Rather than relying on the matrix scheme produced by [AKAO90] or [ZULT90] Lamia proposes a different notation for the labeling of QFD matrices [LAMI95]; this notation will be adopted for the remainder of this dissertation. Lamia simply uses a “A x B” notation where A is the type of artifact/component on the left hand side of the QFD matrix and B is the artifact/component across the top of the matrix (e.g. a Risk to Specification matrix would be labeled “Risk x Specification”, or the “roof” portion of the matrix would be labeled “Specification x Specification”). This leads us to define the relationship between any two sets of components as a QFD Correlation. A QFD Dependency is a particular type of relationship that exists between components. These are formally defined as follows:

Definition 9. (QFD Correlation) Let $H = (N, E, I, O)$ be a relational hypergraph and $C_1 \subset N$ and $C_2 \subset N$ be two sets of components in the hypergraph where

every element of C_1 is of the same type and every element of C_2 is of the same type. If there exists a hyperedge between C_1 and C_2 then the *QFD Correlation* for the QFD matrix $C_1 \times C_2$ is the adjacency matrix $\mathbf{M}_{C_1 \times C_2}$ between C_1 and C_2 such that nonzero values in $\mathbf{M}_{C_1 \times C_2}$ represent the strength of relation between adjacent components.

Definition 10. (QFD Dependency) Let $H = (N, E, I, O)$ be a relational hypergraph and $C = \{c_1, c_2, \dots, c_m\} \subset N$ be a set of components in the hypergraph of the same type. A *QFD Dependency* D is an attribute of C such that for each $1 \leq i \leq m$, $D(c_i) = d_i \in \mathbb{R}$.

Definition 11. (QFD Dependency Deployment) Let $H = (N, E, I, O)$ be a relational hypergraph and $C_1 \subset N$ and $C_2 \subset N$ be two disjoint sets of components in the hypergraph where every element of C_1 is of the same type and every element of C_2 is of the same type. If there exists a hyperedge between C_1 and C_2 and a QFD Dependency $\mathbf{D}_1 = (d_i)$ on C_1 , then the *QFD Dependency Deployment* for the QFD matrix $C_1 \times C_2$ is the vector $\mathbf{D}_2 = (d_j)$ such that $\mathbf{D}_2 = \mathbf{D}_1 \mathbf{M}_{C_1 \times C_2}$ and $\sum_j d_j = \sum_i d_i$.

B. THE MATHEMATICS OF DEPENDENCY DEPLOYMENT

Underlying the deployment of any dependency is a sequence of matrix multiplications that follow from the values of the particular dependency and the values of the correlation matrix associated with the two components that the dependency is to be deployed across.

1. Deployment Equations

First consider a typical "downstream" deployment of dependency from component A to component D as illustrated in the QFD matrix in Table 10.

		D1	D2	D3	D4
		d_1	d_2	d_3	d_4
A1	a_1	b_{11}	b_{12}	b_{13}	b_{14}
A2	a_2	b_{21}	b_{22}	b_{23}	b_{24}
A3	a_3	b_{31}	b_{32}	b_{33}	b_{34}

Table 10 "Downstream" Dependency Deployment

Let \mathbf{A} be a vector of values of a particular QFD Dependency of order m (see Equation 4).

$$\mathbf{A} = [a_1, a_2, \dots, a_m] \quad \text{Equation 4}$$

As expressed in Equation 5, let c_0 be a scalar equal to the sum of those dependencies (in essence this becomes the "amount" of the dependency to be deployed across the design).

$$c_0 = \sum_{i=1}^m a_i \quad \text{Equation 5}$$

Let \mathbf{B} be the QFD Correlation matrix between two sets of components of order $m \times n$ as expressed in Equation 6.

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & & \\ b_{21} & b_{22} & & \\ & & \ddots & \\ & & & b_{mn} \end{bmatrix} \quad \text{Equation 6}$$

As expressed in Equation 7, let \mathbf{E} be the vector result of \mathbf{AB} with order n , and let c_1 be the scalar sum of the values of \mathbf{E} .

$$\mathbf{E} = \mathbf{AB} \quad \text{and} \quad c_1 = \sum_{j=1}^n e_j \quad \text{Equation 7}$$

To maintain a constant value of the dependency during the deployment, \mathbf{D} is the vector result of the dependency deployment and is normalized with respect to c_0 as expressed in Equation 8.

$$\mathbf{D} = \frac{c_0}{c_1} \mathbf{E} \quad \text{and} \quad c_3 = \sum_{j=1}^n d_j = c_0 \quad \text{Equation 8}$$

Note that $c_3 = c_0$ (the "amount" of the dependency remains constant throughout the deployment).

What remains is to determine whether it is possible to deploy a dependency backwards (upstream) through a QFD matrix (i.e. is it possible to find the original \mathbf{A} , given dependencies \mathbf{D} and correlations \mathbf{B} -- see example in Table 11).

		D1	D2	D3	D4
		d_1	d_2	d_3	d_4
A1	a_1	b_{11}	b_{12}	b_{13}	b_{14}
A2	a_2	b_{21}	b_{22}	b_{23}	b_{24}
A3	a_3	b_{31}	b_{32}	b_{33}	b_{34}

Table 11 "Upstream" Dependency Deployment

To investigate whether this is possible, consider the following derivation. Combining Equation 7 and Equation 8 then solving for **A** leads to Equation 9.

$$\mathbf{D} = \frac{c_0}{c_1} \mathbf{A} \mathbf{B} \quad \text{Equation 9}$$

If \mathbf{B}^{-1} exists, rearranging terms and taking advantage of the fact that $c_0 = c_3$, leaves Equation 10.

$$\mathbf{A} = \frac{c_1}{c_3} \mathbf{D} \mathbf{B}^{-1} \quad \text{Equation 10}$$

But there are two problems with Equation 10. First, seldom will **B** be square and invertible (thus our assumption that \mathbf{B}^{-1} exists is likely invalid) and secondly, there is no way to determine the value of c_1 without knowledge of **A** (recall Equation 7).

This second dilemma remains even if a right pseudo-inverse for a non-square **B** is found and used. To illustrate this dilemma, consider the following derivation. By definition let \mathbf{B}_R^+ be the right pseudo-inverse of **B** as shown in Equation 11. Following from Equation 9 leads to Equation 12.

$$\mathbf{B} \mathbf{B}_R^+ = \mathbf{I} \quad \text{Equation 11}$$

$$\mathbf{D} \mathbf{B}_R^+ = \frac{c_0}{c_1} \mathbf{A} \mathbf{B} \mathbf{B}_R^+ \quad \text{Equation 12}$$

Rearranging terms and taking advantage of the fact that $c_0 = c_3$, leaves Equation 13.

$$\mathbf{A} = \frac{c_1}{c_3} \mathbf{D} \mathbf{B}_R^+ \quad \text{Equation 13}$$

The dilemma with finding the inverse of **B** has been solved but the dilemma of not being able to determine the value of c_1 without knowledge of **A** remains.

2. Downstream Dependency Deployment Example

The above relationships for deploying dependency across a QFD matrix can be illustrated with the following small example.

Suppose that the software customer would like to deploy a value of risk associated with requirements across the design to specifications (see Figure 39).



Figure 39 Deployment of Risk Example

Suppose there are three requirements $\{R1, R2, R3\}$ with associated risk values of $\{5, 1, 3\}$ and these are to be deployed across four specifications $\{S1, S2, S3, S4\}$ (see Equation 14).

$$\mathbf{A} = [5 \quad 1 \quad 3] \quad \text{and} \quad c_0 = \sum_{i=1}^m a_i = 9 \quad \text{Equation 14}$$

Further suppose that the associated QFD Correlation matrix between the three requirements and the four specifications is as shown in Equation 15.

$$\mathbf{B} = \begin{bmatrix} 0 & 1 & 0 & 3 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 9 & 0 \end{bmatrix} \quad \text{Equation 15}$$

Thus, the House of Quality prior to the deployment of risk would look as shown in Table 12, where $\mathbf{D} = (d_j)$ is the vector of Specification Risk that is sought.

	Specs	S1	S2	S3	S4
Rqt	Risk	d_1	d_2	d_3	d_4
R1	5	0	1	0	3
R2	1	3	1	0	0
R3	3	0	0	9	0

Table 12 QFD Matrix for Risk Deployment Example

The underlying Hypergraph for this QFD matrix is illustrated in Figure 40 and the underlying weighted digraph for the correlation matrix is illustrated in Figure 41.

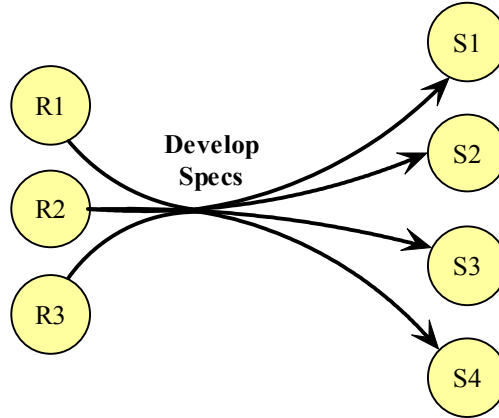


Figure 40 Hypergraph Representation of the QFD Example

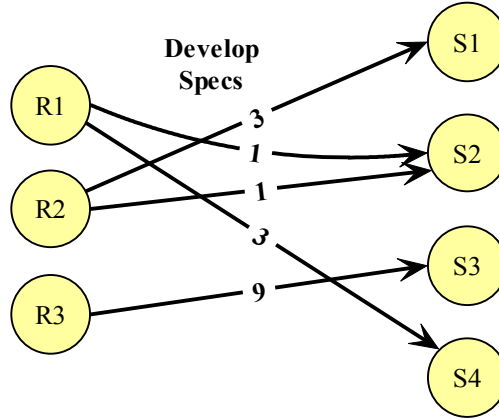


Figure 41 Weighted Digraph Representation of the QFD Example

Using Equation 7 and Equation 8 to calculate **D** (rounded to two decimal places) leads to Equation 16 and Equation 17.

$$\mathbf{E} = \mathbf{AB} = [3 \quad 6 \quad 27 \quad 15]$$

and

$$c_1 = \sum_{j=1}^n e_j = 51$$

Equation 16

$$\mathbf{D} = \frac{c_0}{c_1} \mathbf{E} = \frac{9}{51} \mathbf{E} = [.53 \quad 1.06 \quad 4.76 \quad 2.65]$$

Equation 17

This gives us a clear picture of how the customer's view of risk associated with the three requirements is "deployed" to the four specifications of the design. It intuitively follows that since requirements R1 and R3 had the greatest risk (5 and 3 respectively) and that they map most strongly to specifications S3 and S4 that these two specifications would end up having the greatest risk.

But is it possible to go backwards, e.g. given the specification risk expressed in Equation 17, is it possible to back-out the customer's initial values of risk for the requirements? The answer, of course, is no -- given the conclusion arrived at in Equation 10 and Equation 13. Because of the lack of independence in the matrix product \mathbf{E} , there is no way for us to arrive at a value of 51 for c_1 .

3. Upstream Deployment of Dependency

The implication of this is significant when put in a software engineering context within the HFSE. It means that the initial input values of dependency are anchored to a particular component of origin. The dependency can be deployed forward from this origin as shown above in Equation 4 through Equation 8. However, values downstream from the origin (forward along the path of development) cannot be modified and then reliably deployed backwards through the matrices to the origin. The best that can be achieved is to create a new dependency at that point (the point of dependency modification) and deploy that dependency backward through the design but using the forward calculations expressed in Equation 4 through Equation 8. In order to account for the proper order of \mathbf{D} and \mathbf{B} , the transpose \mathbf{B}^T of the correlation matrix \mathbf{B} is used. For instance, given \mathbf{D} and \mathbf{B} we would arrive at vector \mathbf{A} as shown through Equation 18 and Equation 19.

$$\mathbf{F} = \mathbf{D}\mathbf{B}^T \quad \text{and} \quad c_4 = \sum_{j=1}^n f_j \quad \text{Equation 18}$$

$$\mathbf{A} = \frac{c_3}{c_4} \mathbf{F} \quad \text{and} \quad c_0 = c_3 \quad \text{Equation 19}$$

In our example above, suppose that the values of risk that were arrived at in Equation 17 were actually original risk values determined by the software developer based on his assessment of how the project might fail while implementing those

specifications. The developer then wants to know how that risk deploys backwards (upstream in the development effort) to the customer's requirements so that he can identify the requirements that have the greatest impact on his view of risk. Applying Equation 18 and Equation 19 to **D** and **B** leads to Equation 20, Equation 21, and Equation 22.

$$\mathbf{F} = \mathbf{DB}^T = \begin{bmatrix} .53 & 1.06 & 4.76 & 2.65 \end{bmatrix} \begin{bmatrix} 0 & 3 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 9 \\ 3 & 0 & 0 \end{bmatrix}$$

Equation 20

$$= \begin{bmatrix} 9.01 & 2.65 & 42.84 \end{bmatrix}$$

$$c_4 = \sum_{j=1}^n f_j = 54.5$$

Equation 21

$$\mathbf{A} = \frac{c_3}{c_4} \mathbf{F} = \frac{9}{54.5} \mathbf{F} = \begin{bmatrix} 1.49 & .44 & 7.07 \end{bmatrix}$$

Equation 22

Note that the result from Equation 22 is not at all similar to the vector in Equation 14. From the developer's point of view it is requirement R3 (not requirement R1) that presents the greatest risk (this follows from the fact that specification S3 had such a relatively high value of risk and was so highly coupled to requirement R3).

4. Other Means of Deploying Dependencies: Constant Range

In deriving Equation 8 a key assumption was made that the "amount" of the dependency would remain constant across the deployment. This assumption impacts the deployment by either "thinning-out" the dependency when it is deployed across numerous components (from a few components) (see example in Table 13) or "concentrating" the dependency on just a few components (when deployed from many components) (see example in Table 14).

	Specs	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
Rqt		.41	.48	1.85	1.03	.14	.34	.41	3.08	.41	1.85
R1	5	0	1	0	3	0	1	0	9	0	0
R2	2	3	1	0	0	1	0	3	0	3	0
R3	3	0	0	9	0	0	0	0	0	0	9

Table 13 Dependency Thinning (Dependency = 10)

	Specs	S1	S2	S3
Rqt		6.33	7.85	25.82
R1	5	0	1	0
R2	1	3	1	0
R3	6	0	0	9
R4	2	0	9	0
R5	4	3	0	0
R6	1	9	0	0
R7	7	0	0	3
R8	2	0	3	0
R9	3	0	1	0
R10	9	1	0	3

Table 14 Dependency Concentration (Dependency = 40)

It is possible to use a different initial assumption when deploying the dependency: instead of keeping the "amount" of the dependency constant, the "value range" of each dependency could be kept constant. This would require the deployed dependency to be normalized against the possible range of the dependency.

Let \mathbf{A} be a vector of values of a particular dependency of order m and let dep_{\min} and dep_{\max} be the possible minimum and maximum values of each a_i as expressed in Equation 23.

$$\mathbf{A} = [a_1, a_2, \dots, a_m], \text{ where } \forall i: dep_{\min} \leq a_i \leq dep_{\max} \quad \text{Equation 23}$$

As shown in Equation 24, let \mathbf{B} be a matrix of correlation values between two components of order $m \times n$, and let cor_{\min} and cor_{\max} be the possible (not necessarily actual) minimum and maximum values of each b_{ij} .

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & & \\ b_{21} & b_{22} & & \\ & & \ddots & \\ & & & b_{mn} \end{bmatrix} \text{ such that } \forall i, j : cor_{\min} \leq b_{ij} \leq cor_{\max} \text{ Equation 24}$$

Let \mathbf{E} be the vector result of \mathbf{AB} with order n shown in Equation 26.

$$\begin{aligned} \mathbf{E} &= \mathbf{AB} \\ \text{and} & \\ \forall j : (dep_{\min})(cor_{\min})(m) &\leq e_j \leq (dep_{\max})(cor_{\max})(m) \end{aligned} \text{ Equation 25}$$

To maintain a constant range of values for the dependency during the deployment, as shown in Equation 26 and Equation 27, \mathbf{D} is the vector result of the dependency deployment and is normalized with respect to the minimum and maximum possible values of e_j . These normalized values are added to the minimum dependency value dep_{\min} .

$$\text{Let } \mathbf{G} = (g_j) \text{ such that } \forall j : g_j = dep_{\min} \text{ Equation 26}$$

$$\begin{aligned} \mathbf{D} &= \left[\frac{(dep_{\max} - dep_{\min})}{[(dep_{\max} * cor_{\max}) - (dep_{\min} * cor_{\min})](m)} \right] \\ &\quad * [\mathbf{E} - (dep_{\min})(cor_{\min})(m)] + \mathbf{G} \end{aligned} \text{ Equation 27}$$

$$\text{and } \forall j : dep_{\min} \leq d_j \leq dep_{\max}$$

In general QFD practice (as discussed in Chapter II) QFD correlation values are typically 0:1:3:9, thus cor_{\min} is usually zero. Since Equation 27 does not explicitly require this to be the case, the engineer must select maximum and minimum dependency values and correlation values with care to ensure that $[(dep_{\max} * cor_{\max}) - (dep_{\min} * cor_{\min})] \neq 0$.

To illustrate the use of this form of deployment, consider the application of Equation 23 through Equation 27 to the QFD deployment presented in Table 12. Let \mathbf{A} be a vector of dependency values within the range [1, 7] and order $m=3$ and let \mathbf{B} be matrix of correlations within the range [0, 9]. This leads to the results in Equation 28 through Equation 30.

$$\mathbf{A} = \begin{bmatrix} 5 & 1 & 3 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 & 1 & 0 & 3 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 9 & 0 \end{bmatrix}, \quad \mathbf{G} = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \quad \text{Equation 28}$$

$$dep_{\min} = 1, \quad dep_{\max} = 7, \quad cor_{\min} = 0, \quad cor_{\max} = 9$$

$$\mathbf{E} = \mathbf{AB} = \begin{bmatrix} 3 & 6 & 27 & 15 \end{bmatrix} \quad \text{Equation 29}$$

$$\begin{aligned} \mathbf{D} &= \left[\frac{(dep_{\max} - dep_{\min})}{[(dep_{\max} * cor_{\max}) - (dep_{\min} * cor_{\min})](m)} \right] \\ &\quad * [\mathbf{E} - (dep_{\min})(cor_{\min})(m)] + \mathbf{G} \\ &= \left[\frac{(7-1)}{[(7*9) - (1*0)](3)} \right] [\mathbf{E} - (1*0*3)] + \mathbf{G} \quad \text{Equation 30} \\ &= \frac{6}{189} \mathbf{E} + \mathbf{G} \\ &= \begin{bmatrix} 1.10 & 1.19 & 1.86 & 1.48 \end{bmatrix} \end{aligned}$$

In examining the result of Equation 30 notice that the values did remain in the range $[dep_{\min}, dep_{\max}]$. However, while these values were obtained through a sound mathematical deployment methodology, the usefulness of such numbers in decision support is questionable. These numbers become even more questionable as they are deployed further and further through the design because they will continue to approach dep_{\min} . The reason the values were particularly low (near dep_{\min}) and why they will continue to get lower as they are further deployed is because of the relative sparseness of the correlation matrix \mathbf{B} . Such sparseness is characteristic of QFD correlation matrices and makes the methodology expressed in Equation 23 through Equation 27 much less useful from a decision support perspective than the methodology expressed in Equation 4 through Equation 8.

While it is possible to account for this sparseness by using the values of e_{\max} and e_{\min} (from $\mathbf{E} = (e_j)$) in Equation 27 instead of $(dep_{\max} * cor_{\max})$ and $(dep_{\min} * cor_{\min})$, respectively, another problem is introduced. Consider Equation 31.

$$\mathbf{D} = \left[\frac{(dep_{\max} - dep_{\min})}{(e_{\max} - e_{\min})} \right] [\mathbf{E} - e_{\min}] + \mathbf{G}$$

Equation 31

$$\text{and } \forall j: dep_{\min} \leq d_j \leq dep_{\max}$$

In this deployment scheme the deployed values that lead to \mathbf{D} are normalized such that the highest value of \mathbf{D} assumes the value of dep_{\max} and the lowest values assumes the value of dep_{\min} . The remaining values are spaced appropriately based on their ratio within the range $[dep_{\min}, dep_{\max}]$. The shortcoming with this scheme is that it does not preserve a sense of ratio from the boundaries of the range (i.e. any boundary that should exist is lost in the deployment). A second and more minor shortcoming (since it is unlikely to occur in actual practice) is that if all of the values of \mathbf{E} are the same, then $e_{\max} = e_{\min}$ and the denominator of Equation 31 goes to zero making the result undefined.

Since the reason for the deployment of dependencies is for decision support, the HFSE will rely on the "constant dependency amount" methodology specified in Equation 4 through Equation 8 as opposed to the "constant dependency range" methodology specified in Equation 23 through Equation 27 or the methodology in Equation 31. This still leaves, however, the dilemma of how to make use of the deployed dependencies subject to dependency "thinning" and dependency "concentration" (recall Table 13 and Table 14).

5. Other Mathematical Checks

There are other mathematical checks that can be performed on a QFD matrix to ensure that it is consistent. For instance, it is possible to examine the columns of a QFD matrix to determine if any superfluous artifacts have been created. It is also possible to perform a "coverage" check of the QFD Correlation matrix to provide visibility as to whether the "how" components adequately implement the "what" components of the design.

a. Superfluous Artifact Analysis

The purpose of superfluous artifact analysis is to determine if there are artifacts in the design that are not needed and have been erroneously introduced. This analysis is conducted by looking for discrepancies in column values of the QFD Correlation matrix. A complete column of zero valued correlations is an indication of a

mistake in the design: either a correlation value was mistakenly omitted, or the “how” artifact itself is superfluous to the design. As an example, consider the QFD matrix shown in Table 15 in which the QFD Dependency “Requirements Priority” has been deployed downstream to specifications.

	Specs	S1	S2	S3	S4	S5
Rqts	Priority	1.4	2.8	4.2	0	3.6
R1	5	0	1	0	0	0
R2	1	1	9	9	0	0
R3	2	3	0	0	0	3
R4	4	0	0	3	0	3

Table 15 QFD Coverage Analysis Example

Note that the correlation column of S4 has only zeros for values. Since the overall purpose of the Specifications is to implement the stated requirements, this column of zeros indicates that either the engineer made a mistake in assigning correlation values or that S4 is superfluous to the design. The “0” value of the deployed priority dependency value for S4 suffices as a “red flag” for the engineer. The engineer should question whether or not S4 is needed in the design and if so, which requirements are driving that need.

b. Coverage Analysis

The purpose of performing a QFD coverage analysis is to determine if adequate “hows” have been specified for implementing the “whats” in the QFD process (e.g. have adequate specifications been identified for implementing the requirements). “Adequate” in this case refers to whether the “how” artifacts are sufficient in quantity and quality to fully implement the “what” artifacts. [ZULT92] proposes that the coverage analysis be performed by summing the rows of the QFD Correlation matrix, normalizing the result, and then comparing these normalized values against the relative weights of the “what” components. They should generally be the same. Mismatches are indications that either more (or stronger) “hows” are needed to implement a particular “what” or that there are too many (or too strong) “hows” that implement a “what”.

As an example of coverage analysis, continue to consider the QFD matrix presented previously in Table 15. Summing the correlation values by row and

normalizing, then comparing them side-by-side with normalized values of the Requirements priorities indicates several additional problems; consider the results in Table 16.

	Normalized Rqts Priority Values	Normalized Correlation Row Sums
R1	0.417	0.031
R2	0.083	0.594
R3	0.167	0.187
R4	0.333	0.187

Table 16 Coverage Analysis Example: Side-by-Side Comparison

It is clear that the requirement R1 (the most important requirement) has been severely underrepresented in specifications and that R2 (the least important requirement) has been significantly over-represented as indicated by the large difference in normalized values corresponding to these two requirements. The software engineer should reconsider the correlation values established in Table 15. The engineer might also want to identify additional specifications needed to fully implement R1. Now consider the adjusted QFD matrix in Table 17 and the corresponding side-by-side coverage analysis values in Table 18.

	Specs	S1	S2	S3	S4	S5
Rqts	Priority	0.75	0.64	3.86	4.82	1.93
R1	5	0	1	0	9	0
R2	1	1	1	0	0	0
R3	2	3	0	0	0	3
R4	4	0	0	9	0	3

Table 17 QFD Coverage Analysis Example

	Normalized Rqts Priority Values	Normalized Correlation Row Sums
R1	0.417	0.333
R2	0.083	0.067
R3	0.167	0.200
R4	0.333	0.400

Table 18 Coverage Analysis Example: Side-by-Side Comparison

The side-by-side comparison in Table 18 reveals a much more consistent result. While the qualitative assessment illustrated in the above example demonstrates the coverage analysis concept, the technique can be supplemented with a more sophisticated statistical analysis such as hypothesis testing using a “Paired t Test” [DEVO00]. Although outside the scope of this dissertation, such analysis can give the engineer ever increasing confidence of adequate coverage of downstream components within the software design.

C. MAKING USE OF DEPLOYED DEPENDENCIES

The goal of deploying dependencies across the design is to allow the software engineer the ability to visualize "slices" of the design that have particular meaning. For example, if the engineer has deployed customer risk throughout the design, he may want to do the following:

- Identify all components that have a relatively high value of customer risk ("relatively" being defined by some user specified threshold value).
- After identifying a particular component (say, with high risk) identify all strongly (or weakly, or any) connected components that trace to that component. The strength of connectivity used in the search is defined by the user specified threshold value.

These two techniques are named Dependency Threshold and Component Trace, respectively.

1. Dependency Threshold

Since the range of dependency values varies widely between groups of components (recall dependency "thinning" and "concentrating"), it is generally not sufficient to simply specify a particular value of dependency and then isolate all components within each group of components that have dependency values greater than

or less than that value. However, since each group of components has been normalized within the group, it is possible to isolate those components that are greater than or less than some specified value related to their mean and standard deviation. Let t be the dependency threshold value specified against the mean and standard deviation of each group of components as shown in Equation 32.

$$t = \mu_i \pm x\sigma_i$$

where x is a user specified value and

i identifies a particular set of components.

Equation 32

As an example, consider the QFD deployment example previously presented in Table 12 and shown below in Figure 42 (dependency values of risk are shown in parentheses).

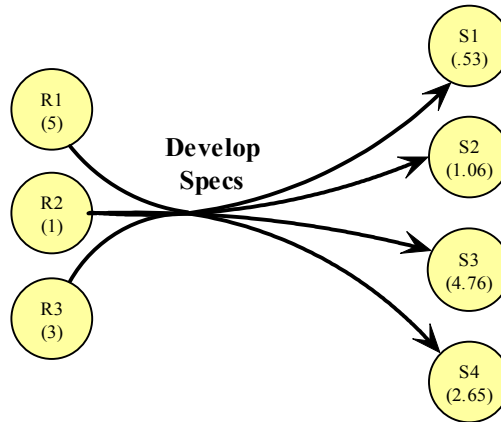


Figure 42 Hypergraph with Risk Dependency Values

The requirements components and specification components have mean and standard deviations as shown in Equation 33.

$$\mu_R = 3 \quad \text{and} \quad \sigma_R \approx 1.63$$

$$\mu_S = 2.25 \quad \text{and} \quad \sigma_S \approx 1.65$$

Equation 33

Applying the "Dependency Threshold" view, Figure 43 illustrates the remaining subgraphs if the underlying hypergraph is trimmed at thresholds of $t = \mu_i$ and $t = \mu_i + \sigma_i$, respectively.

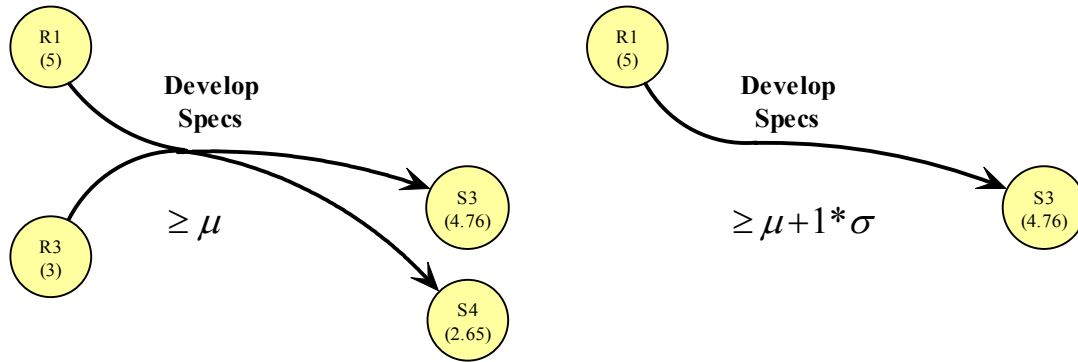


Figure 43 Subgraphs Trimmed with Dependency Threshold

From a decision support perspective, the Dependency Threshold view of the underlying hypergraph allows a software engineer to easily identify particular subgraphs of interest. It is relatively easy to identify those components that have the greatest (or least) dependency values and to base resource allocation decisions upon those views.

2. Component Trace

The Component Trace view of a hypergraph allows the software engineer to identify all connected components of a single component based on a user-defined value of connectedness. This technique assumes that the same ranges of values are used throughout the hypergraph for relating the correlation between any two components. Consider Figure 44, a weighted digraph representing a simple software development process.

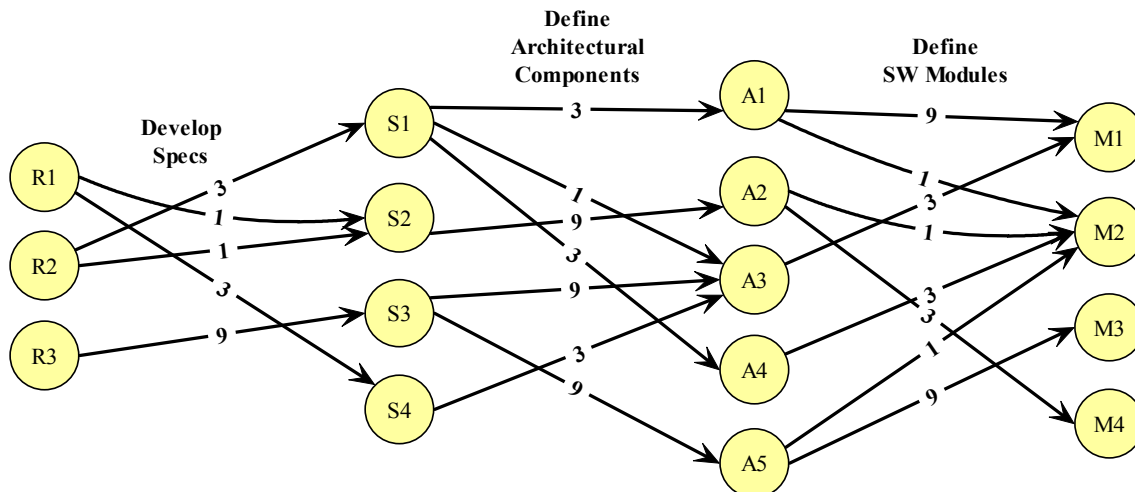


Figure 44 Weighted Digraph Example

Component traces centered on component A3 with threshold values of 2 and 8 are illustrated below in Figure 45 and Figure 46, respectively.

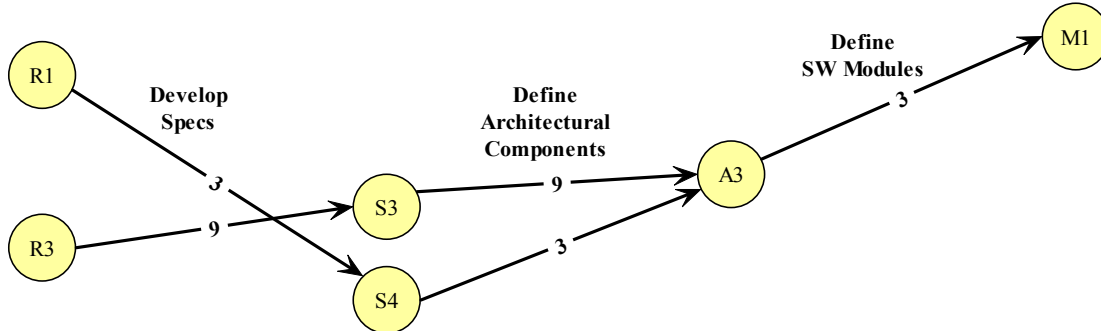


Figure 45 Component Trace from A3 with Threshold 2

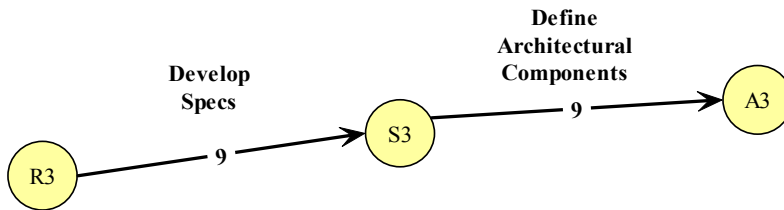


Figure 46 Component Trace from A3 with Threshold 8

Again from a decision support perspective, the Component Trace view of the underlying hypergraph allows a software engineer to easily identify particular subgraphs of interest. Using such a view it is relatively easily to identify the potential knock-on effects associated with changes to a particular component and to base resource allocation decisions upon those potential effects.

D. ESTABLISHING DEPENDENCY VALUES

Central to the use of the QFD methodology within the HFSE is the concept of dependency value deployment. "Setting priorities means advancing some actions and postponing others" [ZULT93]. The design realities implied by this quotation are substantial and while the actual dependency values used by the HFSE are developed in external software development tools and then imported into the HFSE, the adage "garbage in equals garbage out" holds true. Thus, if erroneous dependency values are

imported into the HFSE and then deployed throughout the design, the engineer is likely to make poor engineering decisions stemming from this data. Therefore, the concepts underpinning what separates good dependency values from bad values are worthy of review.

1. Scales of Measurement

In his seminal work, Stevens [STEV46, 51] provides what is now considered to be the standard classification taxonomy for measuring both quantitative and qualitative variables. His taxonomy of measurement consists of four different ways to use numbers to measure everyday phenomena. These four ways of measurement consist of nominal, ordinal, interval, and ratio.

a. Nominal

In nominal measurement, numbers are simply used as labels for identifying separate entities. They denote no additional information other than identification. An example of a nominal measurement would be the random assignment of a student identification number. The number serves no purpose other than to distinguish one student from another and cannot be used to deduce which student is smarter or which student is majoring in chemistry. From an HFSE dependency valuation perspective, nominal values provide no useful information about software artifacts (except to distinguish between artifacts) and thus should never be used as deployable dependency values.

b. Ordinal

In ordinal measurement, entities are simply ordered in a particular way: least to most, best to worst, worst to best, etc. In this case, it is possible to distinguish which entity might be “better” than another, but it is not possible to distinguish how much better. Values from an ordinal scale distinguish the direction of difference only. An example of an ordinal measurement would be the assignment of finishing positions to a set of racehorses completing a race. While it is possible to determine which horse finished ahead of another, an ordinal scheme provides no information that would allow a user to determine by how much one horse beat another. From an HFSE dependency valuation perspective, ordinal values might provide some marginally useful information about software artifacts differences but as discussed later, this information should be used

cautiously and results based on this type of dependency calculation should be viewed skeptically.

c. Interval

In interval measurement, the distance between two entities is established. In this case, it is possible to distinguish which entity is better and by how much using a scale consistent throughout the entire range of the scale. Interval scales do not have true zero points. Temperature as measured in Fahrenheit or Celsius is an example of an interval scale. Using an interval scale for temperature, it is possible to determine which temperature is warmer than another and by how much, however, because of the lack of a zero point it is meaningless to try to state that one temperature is twice as warm as another. From an HFSE dependency valuation perspective, interval values provide useful information about software artifacts and results stemming from interval values lead to more dependable information than those generally achieved using an ordinal scale.

d. Ratio

Just as in the interval scale, in a ratio scale numbers tell the direction and relative distance between entities being measured. Additionally, however, ratio scales enjoy the presence of an absolute zero so that it is possible to compare entities in terms of proportions, percentages, and ratios. Measures of distance and mass are examples of ratio valuations. In measuring length, it is possible to distinguish which length is greater, by how much, and by what percentage (e.g., a yardstick is longer than a ruler, it is longer by two feet, or it is 200% longer). From an HFSE dependency valuation perspective, ratio values provide the best information about differences in the dependencies of software artifacts and results stemming from ratio values lead to the most dependable information.

While Stevens' measurement taxonomy [STEV46, 51] appears to be the most widely used and accepted taxonomy, it is worth mentioning that there are some criticisms of his taxonomy and that other taxonomies, besides his, do exist. Comparison of the deficiencies and potential benefits to the HFSE of all of these taxonomies of measurement is outside the scope of this dissertation; however, interested readers are referred to [VELL93] which provides a broad overview of the issues involved.

2. QFD Dependency Valuation

Following from Stevens' measurement taxonomy [STEV46, 51], [COHE95] discusses three different ways of establishing basic values for the priority values of a set "whats" in the QFD valuation process: by Absolute Importance, by Relative Importance, and by Ordinal Importance. In terms of Stevens' taxonomy, these correspond to interval, ratio, and ordinal, respectively.

a. *Absolute Importance (Interval Valuation)*

In using the Absolute Importance scheme, each dependency value is established based on the user's view as to the value of the dependency compared to a set, fixed, absolute scale. For instance, such a scale might be 1-5 as shown in Table 19.

Value	Criteria
1	Not at all helpful
2	Of minor help
3	Of moderate help
4	Very helpful
5	Extremely helpful

Table 19 Example Absolute Importance Scale

Such values are usually obtained using a survey of the particular users in which the respondents are asked to rate the importance of each item related to the particular dependency under consideration. At first, one might wonder why this scale is not considered a ratio scale since it appears to have an absolute zero. However, while the interval values themselves are fixed, this lower bound is not actually absolute. For example, there is nothing that prevents the user from later establishing a new fixed value, for instance -3 , for "Moderately unhelpful." Such an addition would not affect the prior valuations at all. A problem with this scheme is that users have a tendency to inflate such ratings, viewing each dependency item as of moderate importance or greater [COHE95]. This has the effect of grouping the results and making them more difficult to interpret and find logical breakpoints in the data. Table 20 is an example of applying the Absolute Importance scheme to a set of specifications where the dependency under consideration is "Difficulty of Implementation". In this case the implementers use a scale of 1 to 5 where

1 corresponds to "easy to implement" and 5 corresponds to "extremely difficult to implement."

Specification #	Difficulty
Specification 1.1	1
Specification 1.2	4
Specification 1.3	4
Specification 1.4	2
Specification 1.5	5

Table 20 Absolute Importance Valuation Scheme

Note that in such a scheme, it is possible for multiple components to have the same value (e.g. Specifications 1.2 and 1.3 both have a value of 4).

b. Relative Importance (Ratio Valuation)

In using the Relative Importance scheme (also known as "ratio-scale importance"), each dependency item is placed on a set 100 point or percentage scale. In this method each dependency item is ranked ordered from least important to most important and then the relative difference between the items is established (e.g. this item is twice as important as the previous one, or this item is barely more important than the previous one, etc.). Typical ranges of values are from about 40 to 85 [COHE95]. Table 21 is an example of applying the Relative Importance scheme to the same set of specifications, again where the dependency under consideration is "Difficulty of Implementation". In this case the implementers used a 100 point scale where the higher the value, the more difficult the specification is to implement.

Specification #	Difficulty
Specification 1.1	20
Specification 1.2	43
Specification 1.3	46
Specification 1.4	25
Specification 1.5	72

Table 21 Relative Importance Valuation Scheme

c. Ordinal Importance (Ordinal Valuation)

A third scheme for establishing dependency values is the Ordinal Importance scheme. In this scheme, the dependency items are simply arranged in order from least importance to most important and assigned a value based on that order. For instance, 10 items would receive values of 1 to 10 where 1 is the least important item and 10 is the most important item. The other items would receive an ordinal value based on their order of placement. Table 22 is an example of applying the Ordinal Importance scheme.

Specification #	Difficulty
Specification 1.1	1
Specification 1.2	3
Specification 1.3	4
Specification 1.4	2
Specification 1.5	5

Table 22 Ordinal Importance Valuation Scheme

d. Comparison of Valuation Schemes

Dependency values are multiplied through the QFD Correlation matrix and normalized to create values for the next set of downstream or upstream artifacts. While the initial dependency values themselves may not have been generated using a ratio scale, the fact that they get multiplied by the proportional values of the correlation matrix and then normalized, means the values are used as if they were established using a ratio scheme whether they were or not. Thus, the ratio of the range of potential values becomes important and this range varies by the scheme. [COHE95] provides insight as to the typical range of values for 20 items under evaluation in QFD processes. The Absolute Value scale (with a scale of 1-5) will produce a scheme theoretically between 1-5; but, in practice, this is about 3-5 or a ratio of 1:1.6. The Relative Value scheme generally produces values in the range 40 to 85, a ratio of 1 to 2.1. The ordinal scheme will produce values of 1 to 20, a ratio of 1:20. The ratios for the Absolute Value scale and Relative Value scale remain fairly constant as the number of items being compared increases; however, the ratio for the ordinal scale continues to grow -- imagine how large

this ratio would be with several hundred items. Thus, using an ordinal scheme with many components and deploying values through a correlation matrix will have the effect of significantly over-emphasizing important “whats” and under-emphasizing least important “whats”. Even though the use of ordinal value schemes is widespread and common in QFD practice, their use should be viewed with skepticism.

The HFSE will take advantage of using dependency values established from other software development tools. For instance, Requisite®Pro uses an absolute (interval) scale for requirement priority (low, medium, high), and allows the user to create user defined metrics based on Relative Value (ratio schemes). Traditional Japanese QFD methodologies rely on using simple Absolute scales of 1-5, with increasingly sophisticated methodologies (e.g. pairwise comparison) used to establish the weights. However, there is evidence that even more complex methods are being used such as the Analytic Hierarchy Process (AHP) [ZULT92], which is a methodology based on ratio scales. [AKAO90] also recommends using AHP when additional rigor is required to sort through and establish well-supported priorities upon which to base decisions. While such techniques as AHP require significantly more effort, they provide greater accuracy and support for consistency checking and sensitivity analysis.

In summary, many software development tools produce metrics that can be imported into the HFSE and deployed throughout the development effort. However, the engineer should be cognizant of the type of values being imported because the type of value affects the accuracy and correctness of the decision support information later produced from those values. Ordinal values may provide some useful indications, but the engineer should be skeptical of basing critical or costly decisions upon them. Absolute values (using interval scales) provide much better results and have been reliably used in QFD practice [COHE95]. Relative values (using ratio scales) provide the best and most mathematically sound valuation and results [SAAT80].

3. The Analytic Hierarchy Process (AHP)

a. The Normalized Principal Eigenvector of Priority Values

Saaty presents the Analytic Hierarchy Process (AHP) as a process to establish ratio valuation of a group of items through the use of pairwise comparison [SAAT80]. This is typically accomplished through the creation of a comparison matrix

using a set Absolute Importance scale. Consider Table 23 in which four items (A, B, C, and D) are compared to each other using the Absolute scale provided in Table 24.

Importance	A	B	C	D
A	1	5	6	7
B	1/5	1	4	6
C	1/6	1/4	1	4
D	1/7	1/6	1/4	1

Table 23 Example AHP Comparison Matrix (after [SAAT80])

Value	Meaning
1	Equally Important
3	Weakly more important
5	Strongly more important
7	Very Strongly more important
9	Absolutely more important

Table 24 AHP Comparison Valuation Scheme

Table 23 illustrates the common form of an AHP comparison matrix: a right-hand reciprocal matrix with values on the diagonal equal to 1. Typically, appropriate groups of stakeholders subjectively establish each value above the diagonal; the actual minimum number of comparisons needed to construct the matrix is $n - 1$ since the diagonal values and reciprocal values below the diagonal are forced.

The vector of priority values is derived from the normalized principle eigenvector of the of this comparison matrix. A good approximation of this vector is given by multiplying the n elements of each row of the matrix, taking the n^{th} root of each result, and then normalizing the resulting vector [SAAT80]. Consider Table 25, which shows the approximate result from Table 23 side-by-side with the actual principal eigenvector.

Element	Approximate Result	Actual Normalized Eigenvector
A	0.61	0.61
B	0.24	0.24
C	0.10	0.10
D	0.04	0.05

Table 25 AHP Example Normalized Priority Values

b. Consistency Checking of the Comparison Matrix

[SAAT80] also provides a sound methodology for confirming the consistency of any set of subjectively derived valuations. Because more subjective comparisons are input into the comparison matrix than are mathematically required ($(n^2 - n)/2$ versus $n - 1$), the comparison matrix may become inconsistent and produce inconsistent priority valuations. Saaty's method for confirming consistency depends on comparing the principal eigenvalue (λ_{\max}) to n . The closer λ_{\max} is to n , the more consistent is the comparison matrix. The first step in confirming consistency is to calculate the consistency index (C.I.) shown in Equation 34 [SAAT80].

$$\text{C.I.} = \frac{(\lambda_{\max} - n)}{(n - 1)} \quad \text{Equation 34}$$

The C.I. is then compared to an empirically derived Random Index (R.I.), which is a measure of the possible randomness of a comparison matrix. This comparison is called the Consistency Ratio (C.R.) and is given by Equation 35.

$$\text{C.R.} = \frac{\text{C.I.}}{\text{R.I.}} \quad \text{Equation 35}$$

Consistency Ratios of 0.10 or less are considered acceptable [SAAT80].

c. Hierarchical Clustering to Account for Non-Independence

The comparison matrix in Table 23 assumed that the four items were independent and that subjective evaluations can be independently made between them. However, what happens if the items are not independent of one another? Frequently, software engineers are asked to provide valuations for hierarchical artifacts that are not independent. Consider the small subset of requirements from the CARA Infusion Pump

shown in Table 26 in which pertinent requirements information is located in all three hierarchical levels.

Tag	Requirement Name	Requirement Description
FEAT7	Occlusion Line Monitoring	The CARA will monitor the occlusion lines whenever the pump is plugged in.
FEAT7.1	Occlusion Detected	If an occlusion fault is detected
FEAT7.1.1	Occlusion Display Msg	An appropriate error message should be issued.
FEAT7.1.2	Occlusion Level 1 Alarm	A level 1 alarm should be issued
FEAT7.1.3	Occlusion Terminate AC	If an occlusion is detected while in auto-control, CARA will terminate auto-control

Table 26 Excerpt from CARA Infusion Pump Requirements

In applying the AHP to this particular case, “clusters” are established to form groupings of independent entities. The clusters here might be considered as {FEAT7} and {FEAT7.1, 7.1.1, 7.1.2, 7.1.3}. After establishing the appropriate weightings between these two clusters, the second cluster is considered separately and further clustered and weighted as needed. The weighting of the higher hierarchy is divided among lower level clusters. Consider the more complex example in Figure 47.

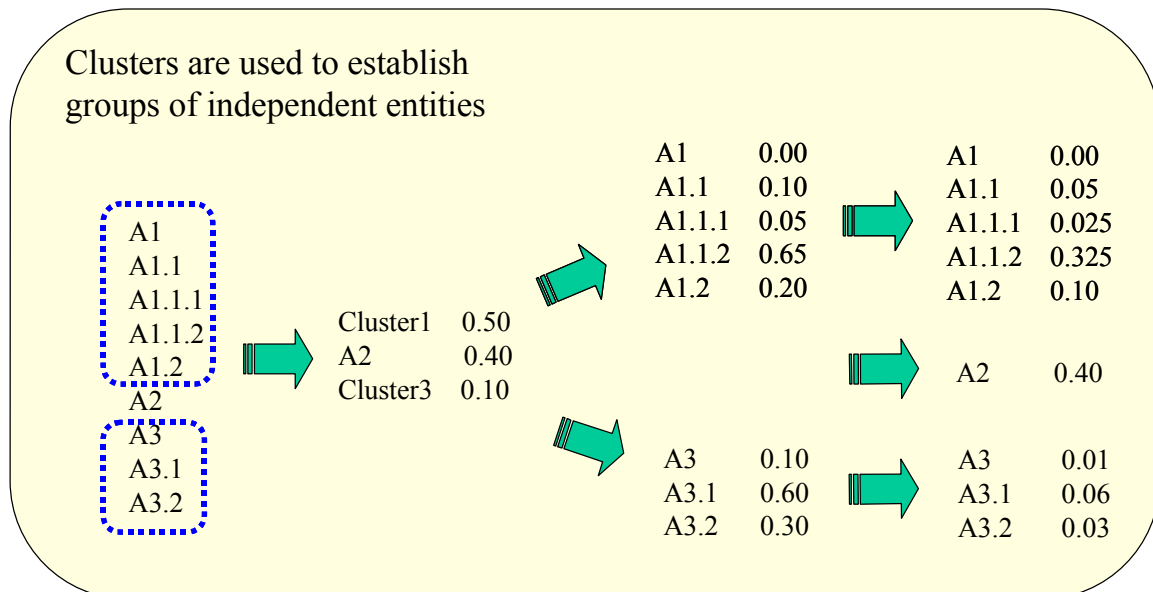


Figure 47 AHP Clustering Example

In this example, nine architectural entities are compared and weighted. Three initial independent clusters are established and weighted (element A2 is considered its own cluster). Clusters 1 and 3 are then separately and internally compared and weighted. While the entities internal to the clusters might not strictly independent, it is generally possible to evaluate the elements in clusters of nine items or less to determine the contribution of each element towards the overall function of the cluster. The weighting of the cluster is then applied to this internal comparison to arrive at a final weighted value for each atomic component. [SAAT80] points out that clustering also serves a second and important purpose and that is to decompose large, complex valuation problems into smaller, more manageable pieces that can be easily understood by the human mind.

4. Subjectivity and Sensitivity Analysis

In any human centric activity requiring judgment, subjectivity is always present. However, it is possible to account for this subjectivity through application of consistent group decision processes and to then confirm results through use of sensitivity analysis. Even though each person perceives the world slightly differently, it is possible to use consensus-building techniques to produce results that are statistically consistent with the overall perceptions of the group. [SAAT80] points out:

...if people do not know what they are talking about, there is no scale that would make them look better. However, if people know something and they want a measure of it, then there is no better way of getting these judgments down than through a systematic procedure which facilitates comparisons, and is in harmony with intuition and human feelings, and is free of artificiality. If a person already knows the answer, he then has no need for any scale...

As discussed in the previous section, AHP is a methodology specifically designed to use groups of stakeholders to perform comparisons and arrive at statistically consistent results. Such methods lend themselves to further evaluation through sensitivity analysis.

The purpose of performing a QFD sensitivity analysis is to determine how sensitive the resulting deployed dependency values are to perturbations in correlation values or in the initial dependency values. While considered an area of future research (see Chapter IX) it should be possible to embed dedicated sensitivity analysis techniques and algorithms into the tool support for the HFSE. Chapter VII discusses the current

status of the tool support of the HFSE. While the tool currently does not contain dedicated sensitivity functionality, the tool in its current state does provide a user the ability undertake “trial-and-error” and “what-if” analysis in support of sensitivity analysis of derived results. Such analysis allows users to identify, quantify, and understand the bounds and limitations of the results provided by the HFSE.

E. CHAPTER SUMMARY

This chapter focused on identifying and discussing the key mathematical constructs needed to represent the HFSE artifacts and perform QFD operations upon them. The extensions to the RH model were defined and the mathematics supporting QFD dependency deployment were presented and illustrated with examples. Alternative deployment schemes were explored and explained. Mathematical techniques for identifying, isolating, and viewing particular subgraphs of interest (induced from the overall underlying hypergraph of the development effort) were presented. Dependency valuation schemes were offered and compared. Finally, the topics of consistency and subjectivity were discussed and techniques for dealing with them (e.g. Coverage Analysis, the Analytic Hierarchy Process, Sensitivity Analysis, etc.) were provided.

V. APPLICATION OF THE OBJECT-ORIENTED METHODOLOGY FOR INTEROPERABILITY TO THE DOMAIN OF SOFTWARE DEVELOPMENT TOOLS

A. CHAPTER OVERVIEW

In Chapter III, the underlying ontologies needed for establishing a Federation Interoperability Object Model (FIOM) were developed. In this chapter, the focus is to explain how the FIOM is actually constructed using those ontologies. Section B describes the motivation and methodology. Section C illustrates the methodology through the use of an example from the software development tool domain. Section D discusses the steps that would be needed in order to add additional tools and development artifacts (such as Rational Rose and pseudo-code) to an existing software development tool FIOM. Section E identifies the remaining tasks that would need to be accomplished in order to actually embed the generated translators from the OOMI IDE into software development tool application add-ons. Section F points out some limitations and challenges of the OOMI approach.

B. BUILDING A FEDERATION INTEROPERABILITY OBJECT MODEL

1. Motivation for the FIOM

[YOUN02b] defines the Object-Oriented Model for Interoperability (OOMI), which relies on establishing a collection of objects that represent real-world entities to provide interoperability between a specific set of heterogeneous systems. This collection of objects is called a Federation Interoperability Object Model (FIOM). The OOMI approach using the FIOM overcomes numerous challenges presented by the differences in modeling between systems. [YOUN02b] provides a classification of these modeling differences, paraphrased as follows:

- Heterogeneity of Hardware and Operating Systems: differences in the hardware and operating system platforms encountered when integrating autonomously developed systems.
- Heterogeneity of Organizational Models: differences in the conceptual models used by autonomously developed systems; dissimilarities in the database models used, such as network, hierarchical, relational, universal, or object structured.

- Heterogeneity of Structure: differences in structural composition, possible schema mismatches, and variations due to the presence of implied information.
- Heterogeneity of Presentation: differences due to domain mismatch problems, the use of different units of measure, differences in precision, disparate data types, and different field lengths or variations in integrity constraints.
- Heterogeneity of Meaning: differences arising from the imprecise nature of natural language for characterizing real-world entities.
- Heterogeneity of Scope: differences that arise from different perspectives on what attributes a given application needs to capture about the real-world entity being modeled.
- Heterogeneity of Level of Abstraction: differences in the level and degree of aggregation of atomic data elements.
- Heterogeneity of Temporal Validity: differences in the time used by two models to observe or record the state of real-world entities.

These modeling differences exist in heterogeneous systems of many domains. Even though the OOMI was originally validated using an example from military C4I systems, this same model can be applied to a different domain -- the domain of software development tools and models.

The OOMI was developed because of numerous limitations in current approaches to interoperability that failed to account for the many differences in modeling. [YOUN02b] identifies these limitations as follows:

First, [traditional approaches to interoperability] do not provide a means for resolving the complete spectrum of modeling differences found among heterogeneous systems. Second, they do not provide assistance in determining when different system models refer to the same entity from the problem domain. Third, in order to access another component or system's state or exercise its behavior, most current approaches require the requesting system to utilize the provider system's model of its state or behavior to access its information... Fourth, most approaches utilize a direct point-to-point conversion process for resolving modeling difference among systems vice a two-step conversion process using an intermediate model... Fifth, most approaches provide no or limited support to development of the translations required to resolve modeling differences among systems. Finally, most approaches are concerned only with the resolution of modeling differences for information exchanged among systems and do not provide the capability for resolving possible differences in the signatures used to access the behavior of corresponding methods on different systems.

The FIOM provides the mechanism for subordinate model interaction in the HFSE. Translators generated using the FIOM provide run-time interaction between subordinate models.

2. FIOM Construction Methodology

The construction methodology used to build and use a Software Development Tool FIOM is illustrated in Figure 48.

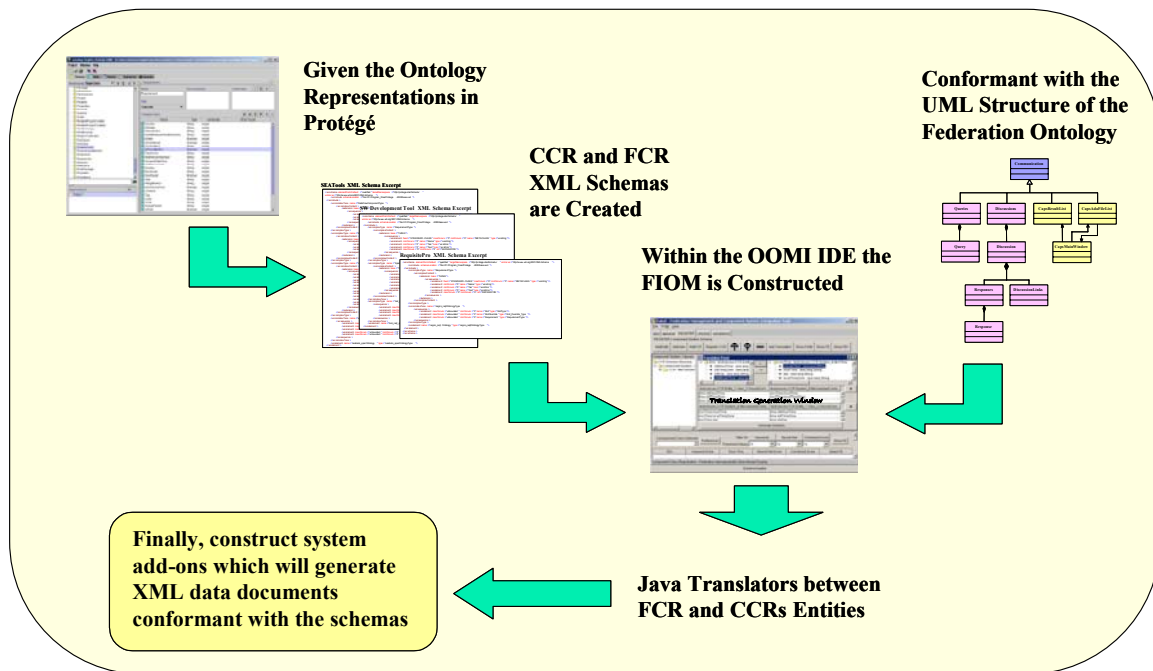


Figure 48 FIOM Construction Methodology

First, as discussed Chapter III, the separate tool ontologies and the federation ontology (of the software development tool domain) are created and represented in Protégé. From Protégé, these ontologies are automatically exported as XML schemas. The individual classes in the tool ontologies are exported as Component Class Representations (CCRs) and the classes in the overarching software development tool ontology are exported as Federation Class Representations (FCRs).

Next, there are two manual interventions required before the XML schemas are imported into the OOMI IDE. First, each of these schemas must be validated against the XML standard used within the OOMI IDE. During this research, these standards were not the same (the version of Protégé used in the research produced schemas that were not compliant with the current W3C standard); so, manual

modifications were made to the schemas using an XML editor (such as XMLSPY® from Altova, Incorporated)[LAWL03]. It should be noted that the need for this manual workaround may be soon eliminated because Protégé users continually provide enhanced and corrected plug-ins based on changing data standards. The second intervention is to modify the OOMI IDE source code. Unfortunately, the OOMI IDE is not yet able to directly import an FCR Schema. Instead, FCRs are currently hard-coded within the OOMI IDE source code, which means the source code itself must be modified to add or change existing FCRs. This is accomplished by importing the validated FCR schema into the OOMI IDE and treating it as a CCR by “Loading” it and “Compiling” it. Save the compiled classes from that schema in the OOMI IDE directory specified in the IDE’s *createMayTestFIOM()* method. The OOMI IDE is then rebuilt and run. After completing these two manual interventions, the remaining tool ontology XML schemas (the CCRs) are imported into the OOMI IDE.

The third step in FIOM construction is to follow the methodology provided in Appendix A of [YOUN02b]. This methodology walks the user through how to use the OOMI IDE to create Federation Entities (FEs) when no FEs exist and how to modify FEs when additional CCRs are added. The creation of the FEs is accomplished based on conforming to the UML inter-relationship diagrams (from Chapter III). The collected set of all FEs constitute the FIOM. Once the FIOM is created, the OOMI IDE provides sets of java skeletons (as incomplete java classes), which provide the structure for the interoperability translators that perform the translation between CCRs and FCRs. [YOUN02b] proposes using reusable component libraries to populate the holes in the translation skeletons. These libraries have yet to be assembled; therefore, it is left to the interoperability engineer to provide the missing code for the translators.

The final step in this process is to construct tool add-ons and embed the translators within these add-ons. The add-ons provide the interface between the tool and the middleware used for data transport throughout the HFSE. Currently, there is no automated method for constructing these add-ons or embedding the translators within them. Within the add-ons, tool specific information is translated, marshaled, and un-marshaled based on the interoperability needs of the tools within the framework.

C. EXAMPLE OF CONSTRUCTING THE FIOM

This section provides an example of the FIOM construction process related to the HFSE. Interested readers are referred to Appendix A of [YOUN02b] for additional details in the methodology in constructing a FIOM. First, the FCR and CCR schemas are automatically exported from Protégé. When generating XML schemas, Protégé provides a “Base” Schema that is common to all Protégé generated schemas. The base schema defines the Protégé namespace. The complete base schema is illustrated in Figure 49.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://protege.stanford.edu/" xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:complexType name="THING">
    <xs:sequence/>
    <xs:attribute name="id" type="xs:ID"/>
  </xs:complexType>
  <xs:complexType name="STANDARD-CLASS">
    <xs:sequence/>
    <xs:attribute name="id" type="xs:ID"/>
  </xs:complexType>
  <xs:complexType name="SlotType">
    <xs:sequence>
      <xs:element name="ASSOCIATED-FACET" type="Instance" minOccurs="0"/>
      <xs:element name="DIRECT-SLOTS" type="Instance" minOccurs="0" maxOccurs="1"/>
      <xs:element name="DIRECT-SUPERSLOTS" type="Instance" minOccurs="0" maxOccurs="1"/>
      <xs:element name="DIRECT-TYPE" type="Class" minOccurs="0"/>
      <xs:element name="DOCUMENTATION" type="xs:string" minOccurs="0" maxOccurs="1"/>
      <xs:element name="NAME" type="xs:string" minOccurs="0"/>
      <xs:element name="SLOT-CONSTRAINTS" type="Instance" minOccurs="0" maxOccurs="1"/>
      <xs:element name="SLOT-DEFAULTS" type="xs:anyType" minOccurs="0" maxOccurs="1"/>
      <xs:element name="SLOT-INVERSE" type="Instance" minOccurs="0"/>
      <xs:element name="SLOT-MAXIMUM-CARDINALITY" type="xs:integer" minOccurs="0"/>
      <xs:element name="SLOT-MINIMUM-CARDINALITY" type="xs:integer" minOccurs="0"/>
      <xs:element name="SLOT-NUMERIC-MAXIMUM" type="xs:float" minOccurs="0"/>
      <xs:element name="SLOT-NUMERIC-MINIMUM" type="xs:float" minOccurs="0"/>
      <xs:element name="SLOT-VALUE-TYPE" type="xs:anyType" minOccurs="0" maxOccurs="1"/>
      <xs:element name="SLOT-VALUES" type="xs:anyType" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="Slot" type="SlotType"/>
  <xs:complexType name="Slot_Override_Type">
    <xs:sequence>
      <xs:element name="ClassName" type="xs:string"/>
      <xs:element name="SlotName" type="xs:string"/>
      <xs:element name="FacetName" type="xs:string"/>
      <xs:element name="Value" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="OSValue">
    <xs:sequence>
      <xs:element name="ClassName" type="xs:string"/>
      <xs:element name="SlotName" type="xs:string"/>
      <xs:element/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="Symbol">
    <xs:restriction base="xs:string"/>
  </xs:simpleType>
  <xs:simpleType name="Instance">
    <xs:restriction base="xs:string"/>
  </xs:simpleType>
  <xs:simpleType name="Class">
    <xs:restriction base="xs:string"/>
  </xs:simpleType>
  <xs:element name="SEPARATOR" type="xs:string"/>
</xs:schema>
```

Figure 49 Protégé Base XML Schema

The Protégé Base XML Schema defines the metadata for the Protégé knowledge base; notice the use of classes, slots, and facets.

The next item to be automatically generated from Protégé is the FCR schema representation of the Software Development Tool Ontology. Recall from Chapter III (Figure 34) that the entire software development tool ontology produced for this HFSE research consisted of approximately 40 classes. The XML Schema in Figure 50 is one small excerpt of that ontology and includes only two classes (i.e. the Requirement class and Specification class).

```
<xs:schema elementFormDefault="qualified" targetNamespace="http://protege.stanford.edu/"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="file:///C:/Program_Files/Protege-2000/base.xsd">
  </xs:include>
  <xs:complexType name="RequirementType">
    <xs:complexContent>
      <extension base="THING">
        <xs:sequence>
          <xs:element fixed="STANDARD-CLASS" maxOccurs="0" minOccurs="0" name="METAClass" type="xs:string"/>
          <xs:element minOccurs="0" name="Name" type="xs:string"/>
          <xs:element minOccurs="0" name="Tag" type="xs:string"/>
          <xs:element minOccurs="0" name="Text" type="xs:string"/>
          <xs:element maxOccurs="0" minOccurs="0" ref="SEPARATOR"/>
        </xs:sequence>
      </extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="SpecificationType">
    <xs:complexContent>
      <extension base="THING">
        <xs:sequence>
          <xs:element fixed="STANDARD-CLASS" maxOccurs="0" minOccurs="0" name="METAClass" type="xs:string"/>
          <xs:element minOccurs="0" name="Name" type="xs:string"/>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="Required_By" type="xs:string"/>
          <xs:element minOccurs="0" name="Tag" type="xs:string"/>
          <xs:element minOccurs="0" name="Text" type="xs:string"/>
          <xs:element maxOccurs="0" minOccurs="0" ref="SEPARATOR"/>
        </xs:sequence>
      </extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="tool_req_specOntologyType">
    <xs:sequence>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="Slot" type="SlotType"/>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="SlotOverride" type="Slot_Override_Type"/>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="Requirement" type="RequirementType"/>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="Specification" type="SpecificationType"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="tool_req_specOntology" type="tool_req_specOntologyType">
  </xs:element>
</xs:schema>
```

Figure 50 Software Development Tool XML Schema Excerpt

As discussed above in Section B, this FCR schema must be manually manipulated so that it can be validated against the current XML standard [LAWL03]. It is imported into the OOMI IDE, treated as a CCR to generate compiled java classes of the schema, and then embedded within the OOMI IDE as an FCR. The OOMI IDE itself is then rebuilt and rerun before importing the tool CCRs.

The tool CCRs are also automatically generated from Protégé and as in the case of the FCR, are manually updated to become compliant with the current XML standard. Figure 51 and Figure 52 illustrate small excerpts of the complete XML schemas from the SEATools ontology and Requisite®Pro ontology, respectively.

```
<xs:schema elementFormDefault="qualified" targetNamespace="http://protege.stanford.edu/"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="file:///C:/Program_Files/Protege-2000/base.xsd">
  </xs:include>
  <xs:complexType name="DataFlowComponentType">
    <xs:complexContent>
      <extension base="THING">
        <xs:sequence>
          <xs:element fixed="STANDARD-CLASS" maxOccurs="0" minOccurs="0" name="METAClass" type="xs:string"/>
          <xs:element minOccurs="0" name="label" type="xs:string"/>
          <xs:element maxOccurs="0" minOccurs="0" ref="SEPARATOR"/>
        </xs:sequence>
      </extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="EdgeType">
    <xs:complexContent>
      <extension base="DataFlowComponentType">
        <xs:sequence>
          <xs:element fixed="STANDARD-CLASS" maxOccurs="0" minOccurs="0" name="METAClass" type="xs:string"/>
          <xs:element minOccurs="0" name="edgeID" type="xs:integer"/>
          <xs:element maxOccurs="0" minOccurs="0" ref="SEPARATOR"/>
        </xs:sequence>
      </extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="VertexType">
    <xs:complexContent>
      <extension base="DataFlowComponentType">
        <xs:sequence>
          <xs:element fixed="STANDARD-CLASS" maxOccurs="0" minOccurs="0" name="METAClass" type="xs:string"/>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="finishWithinReqmts" type="xs:string"/>
          <xs:element minOccurs="0" name="formalDesc" type="xs:string"/>
          <xs:element minOccurs="0" name="informalDesc" type="xs:string"/>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="mcpReqmts" type="xs:string"/>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="metReqmts" type="xs:string"/>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="mrtReqmts" type="xs:string"/>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="periodReqmts" type="xs:string"/>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="specReqmts" type="xs:string"/>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="triggerReqmts" type="xs:string"/>
          <xs:element minOccurs="0" name="vertexID" type="xs:integer"/>
          <xs:element maxOccurs="0" minOccurs="0" ref="SEPARATOR"/>
        </xs:sequence>
      </extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="seatools_specOntologyType">
    <xs:sequence>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="Slot" type="SlotType"/>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="SlotOverride" type="SlotOverride_Type"/>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="DataFlowComponent" type="DataFlowComponentType"/>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="Edge" type="EdgeType"/>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="Vertex" type="VertexType"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="seatools_specOntology" type="seatools_specOntologyType">
  </xs:element>
```

Figure 51 SEATools XML Schema Excerpt

```

<xs:schema elementFormDefault="qualified" targetNamespace="http://protege.stanford.edu/" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="file:///C:/Program_Files/Protege-2000/base.xsd">
  </xs:include>
  <xs:complexType name="RequirementType">
    <xs:complexContent>
      <extension base="THING">
        <xs:sequence>
          <xs:element fixed="STANDARD-CLASS" maxOccurs="0" minOccurs="0" name="METAClass" type="xs:string"/>
          <xs:element minOccurs="0" name="Name" type="xs:string"/>
          <xs:element minOccurs="0" name="Tag" type="xs:string"/>
          <xs:element minOccurs="0" name="Text" type="xs:string"/>
          <xs:element maxOccurs="0" minOccurs="0" ref="SEPARATOR"/>
        </xs:sequence>
      </extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="reqpro_reqtOntologyType">
    <xs:sequence>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="Slot" type="SlotType"/>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="SlotOverride" type="Slot_Override_Type"/>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="Requirement" type="RequirementType"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="reqpro_reqtOntology" type="reqpro_reqtOntologyType">
  </xs:element>
</xs:schema>
</xs:schema>

```

Figure 52 Requisite®Pro XML Schema Excerpt

After importing the CCRs into the OOMI IDE, the FIOM is constructed by defining Federation Entity Views (FEVs) and Federation Entities (FEs). FEVs establish the one to one relationship between a particular FCR and CCR pair. FEs are groupings of FEVs that represent the complete set of representations of the same real-world entity between all the systems in a federation. Figure 53 illustrates the “requirement” Federation Entity for the software tool FIOM consistent with the imported FCR and CCR XML Schemas in Figure 50 and Figure 52, respectively.

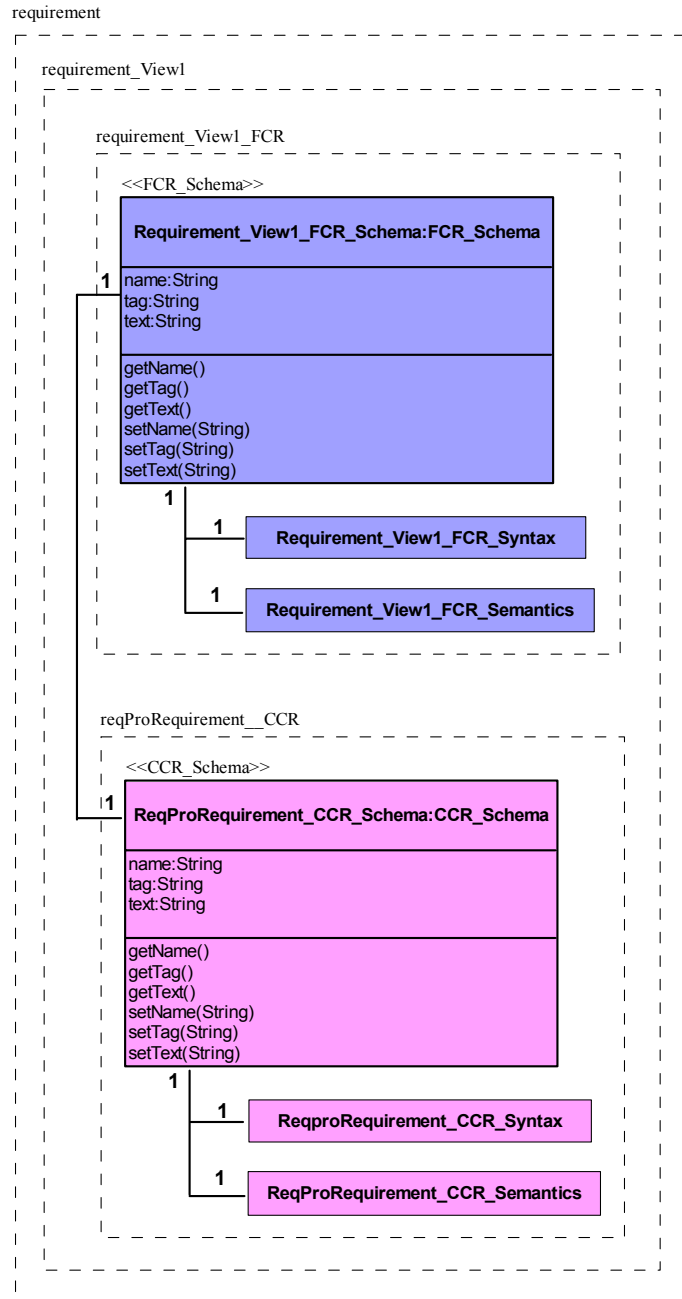


Figure 53 Requirement Federation Entity (FE)

Note that the class representation in the CCR and FCR are identical. As [YOUN02b] points out, this is typical in the FIOM construction process; the initial FCR in an FE is constructed so that it mimics the first CCR that is introduced in the model. The complete set of FEs form the FIOM.

D. EXTENDING THE FIOM TO ACCOUNT FOR ADDITIONAL TOOLS

Given an established FIOM, an important question to be answered is how this FIOM is then modified and extended when additional tools are added to the framework. Such extensions are accomplished by undertaking the ontology addition/modification process defined in Chapter III (end of Section B) and then modifying the FIOM in accordance with [YOUN02b], Appendix A. An important concept during this process is to establish a balance between the granularity available in the tool description of the software artifact and the desired granularity of the set of holistic relationships required from the HFSE. As an example to illustrate the issues involved in such extensions, consider the possible addition of Rational Rose® and a set of pseudo-code to an already constructed FIOM in the HFSE.

1. Addition of Rational Rose® Example

Given an already established HFSE FIOM (perhaps consisting of SEATools and Requisite®Pro objects) adding the UML modeling tool, Rational Rose, requires a number of steps. First, using the methodology of Chapter III, an ontology is created that defines the Rose artifacts to be represented within the HFSE. In the case of Rose, the engineer must determine what level of artifact granularity is available and what level of granularity is desired within the HFSE. Rose provides a COM extensibility interface that allows the engineer access to objects within a Rose model. The level of granularity varies between the entire model itself (a single .mdl file) and the elements within that model (e.g. the object representations of the use case actors, model views, class diagram classes, sequence diagram activities, etc.). For the sake of this example, assume that the engineer wishes to reference the specific elements of a Rose model (as opposed to simply the model itself). The user accesses these “Model Elements” through the Rose Extensibility Interface (see Figure 54).

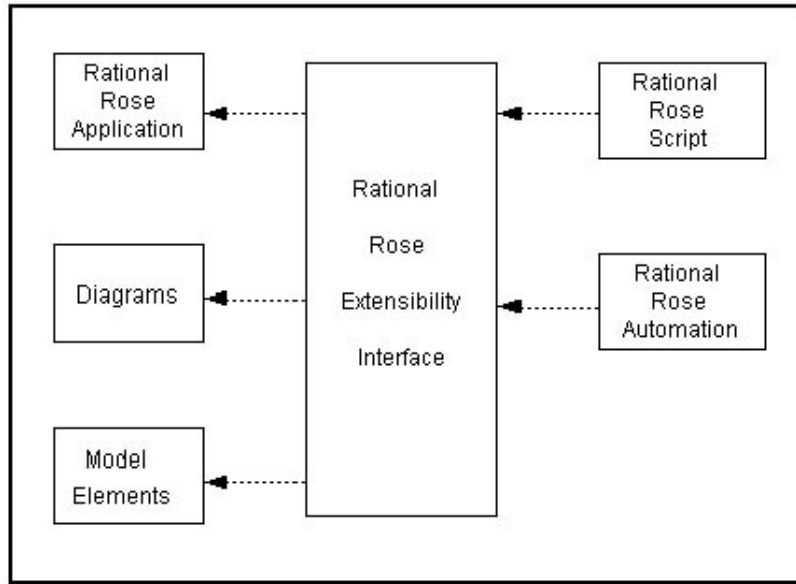


Figure 54 Rational Rose Extensibility Interface (from [ROSE02])

After the user creates an ontology and UML representation of the Rose elements, he must then establish how this UML representation interacts with the already existing UML defined relationships of the FIOM. The FIOM UML description is then modified to account for the additional objects in the federation. Just modifying the UML description is not of course sufficient; the FIOM itself must be modified. This is accomplished by exporting the XML schema related to the Rose objects (the Rose CCR), importing that CCR into the OOMI IDE, building new FEVs to account for the new CCR, and then modifying the relationships between FEVs to update the FEs [YOUN02b]. At that point, the OOMI IDE generates new translators that are then re-embedded into all the tool add-ons throughout the HFSE. Also, the engineer must construct an add-on for Rose so that it can interact with the other tools in the HFSE.

2. Addition of Pseudo-Code Example

Now consider a second example of wanting to include portions of pseudo-code into the HFSE. The issue associated with granularity becomes particularly problematic. Also problematic is how the artifacts are stored and accessed by the HFSE. Assume that the user has defined several algorithms using pseudo-code along the lines of those shown in Figure 55 and Figure 56.

```

INSERTION-SORT(A)
1  for j  $\leftarrow$  2 to length[A]
2      do key  $\leftarrow$  A[j]
3           $\triangleright$  Insert A[j] into the sorted sequence A[1..j - 1].
4          i  $\leftarrow$  j - 1
5          while i > 0 and A[j] > key
6              do A[i + 1]  $\leftarrow$  A[i]
7                  i  $\leftarrow$  i - 1
8          A[i + 1]  $\leftarrow$  key

```

Figure 55 Insertion-Sort Algorithm Pseudo-Code (from [CORM91])

```

MERGE-SORT(A, p, r)
1  if p < r
2      then q  $\leftarrow$   $\lfloor (p + r) / 2 \rfloor$ 
3          Merge - Sort(A, p, q)
4          Merge - Sort(A, q + 1, r)
5          Merge(A, p, q, r)

```

Figure 56 Merge-Sort Algorithm Pseudo-Code (from [CORM91])

The issue now becomes how these software development artifacts (pseudo-code algorithms) are created, stored, and maintained. Assuming a worst case in which the pieces of code are simply scribbled on separate pieces of paper, the engineer must artificially create an electronic description of these artifacts that can be exploited by the HFSE. This artificial description might be as simple as an electronic CSV file that lists the two major artifacts (see Table 27).

ID	Name	Description
IS	Insertion-Sort Algorithm	An algorithm that uses insertion to sort items. Efficient for a small number of elements.
MS	Merge-Sort Algorithm	A recursive algorithm that employs a divide-and-conquer approach to sorting.

Table 27 Large Granularity Pseudo-Code CSV File

Again, granularity becomes an issue. The engineer might need to specify the artifacts to a finer level of granularity in order to link them appropriately to other objects in the

HFSE (e.g. to specific lines of code). In such a case, the CSV file might look like the one in Table 28.

ID	Name	Description
IS	Insertion-Sort Algorithm	An algorithm that uses insertion to sort items. Efficient for a small number of elements.
IS1	Insertion-Sort Line 1	for $j \leftarrow 2$ to $length[A]$
IS2	Insertion-Sort Line 2	do $key \leftarrow A[j]$
IS3	Insertion-Sort Line 3	\triangleright Insert $A[j]$ into the sorted sequence $A[1..j-1]$.
IS4	Insertion-Sort Line 4	$i \leftarrow j-1$
IS5	Insertion-Sort Line 5	while $i > 0$ and $A[j] > key$
IS6	Insertion-Sort Line 6	do $A[i+1] \leftarrow A[i]$
IS7	Insertion-Sort Line 7	$i \leftarrow i-1$
IS8	Insertion-Sort Line 8	$A[i+1] \leftarrow key$
MS	Merge-Sort Algorithm	A recursive algorithm that employs a divide-and-conquer approach to sorting.
MS1	Merge-Sort Line 1	if $p < r$
MS2	Merge-Sort Line 2	then $q \leftarrow \lfloor (p+r)/2 \rfloor$
MS3	Merge-Sort Line 3	Merge - Sort (A, p, q)
MS4	Merge-Sort Line 4	Merge - Sort ($A, q+1, r$)
MS5	Merge-Sort Line 5	Merge (A, p, q, r)

Table 28 Fine Granularity Pseudo-Code CSV File (after [CORM91])

The amount of effort that the engineer should spend to artificially create an electronic description of the artifacts must be carefully weighed against the potential payoff in long-term accessibility, visibility, and maintainability provided by the electronic description as well as the potential pay-off associated with having these artifacts linked through QFD dependencies to the other artifacts in the HFSE.

Once an electronic representation of the artifacts is established, the engineer is then free to add them to the HFSE just as he would any other artifacts: create an ontology description of the artifacts, export that description as an XML schema, update the FIOM, generate new translators, create an add-on that interfaces the pseudo-code CSV file, and re-embed the translators in the application add-ons. One issue remains, however, and that

is that the engineer must manually (and continually during the entire lifecycle) maintain proper configuration management between the electronic description of the artifacts and the real artifacts scribbled on pieces of paper.

E. CREATING THE TOOL ADD-ONS

The final step in applying the OOMI methodology to the HFSE is to construct tool add-ons and embed the translators within these add-ons. These add-ons provide the interface between the tool and the middleware used for data transport throughout the HFSE. Neither this dissertation research, nor the research presented in [YOUN02b] validated this major step of the OOMI approach. While it does not appear to be particularly problematic, this step does represent additional overhead when adopting the OOMI approach within the HFSE. Considerations in constructing the add-ons include the following:

- The add-on must interface with both the tool API and the HFSE middleware mechanisms.
- The translators within the add-ons should be easy to replace and update.
- The tool add-on should be constructed in such a way as to anticipate future upgrades of the tool while minimizing potential changes to the add-on.

Validating this step of the research is considered an area of future research (see Chapter IX).

F. LIMITATIONS OF THE OOMI APPROACH TO PROVIDING INTEROPERABILITY WITHIN THE HFSE

A significant advantage of the OOMI approach is that it reduces the number of potential translations needed between heterogeneous systems ($2n$ versus n^2). This is an important consideration given the number of potential tools that a developer might use over the lifetime of a software development effort. Another advantage is the incremental nature of the approach; the developer can select how many (or how few) different tools to include in the HFSE and how many (or how few) artifacts to include. However, despite these advantages there are a number of limitations in the approach that should be considered.

1. The Intra-lingual Concept

In previous natural language translation research, intra-lingual approaches have been attempted without success. The reasons for these failures inevitably stemmed from the size and the complexity of the task. It was just too difficult to establish an intra-lingua that could account for all of the semantic differences posed by different culturally influenced languages. Having pointed out this limitation, there is some reason to believe that establishing an intra-lingua for heterogeneous computer systems might be more successful. These reasons include the more manageable size and complexity of the problem and the existence of tighter bounds on the semantic differences between computer systems than those that exist in culturally defined natural languages. However, it remains to be validated whether the approach is manageable when the number of heterogeneous systems and artifacts continues to increase.

2. Scalability

As previously discussed, adding additional tools to an ontology or a FIOM is not a simple additive process. There is a lot of work that must take place to reorganize the internal structure of the ontology and FIOM to accommodate new tools. The question then becomes one of scalability. At what point in the process does the internal reorganization required to accommodate a new tool make the approach to become infeasible? This issue is considered an issue for future research (see Chapter IX).

3. Ontologies and FIOMs are Difficult to Build

Ontologies are hard to build, especially in an automated way. Construction takes a great deal of time and effort because the same issues of understandability, identified in this dissertation as motivation for the HFSE, exist in the construction of ontologies and later in construction of the FIOM. It is difficult and time consuming to understand and comprehend the previous efforts of an ontology/FIOM engineer before making significant progress in extending an ontology or FIOM. While Young's OOMI model is an attempt to represent ontologies in an incremental and computationally useful way, addition work is needed to make the overall approach more pragmatic.

G. SUMMARY

This chapter illustrates through the use of examples how a HFSE FIOM is constructed and used given ontology descriptions of HFSE artifacts. The chapter also provides an overview of the tasks required when adding additional tools to an HFSE FIOM and discusses the limitations of the approach.

VI. THE HOLISTIC FRAMEWORK FOR SOFTWARE ENGINEERING (HFSE)

A. THE HFSE

The HFSE is both a methodology and a model (with tool support) for integrating two or more software development tools so that they provide a holistic view of software development artifacts created and used during a software development lifecycle. This chapter explains how to apply the HFSE to integrate a set of software development tools, how to extend the HFSE to include additional tools, and how to use the HFSE to obtain user defined holistic views of the software development process.

B. APPLYING THE HFSE TO ESTABLISH INTEROPERABILITY OF SOFTWARE ENGINEERING PROCESS MODELS

1. Identifying The Project Schema

The first step in applying the HFSE is to identify the project schema of the particular software development effort. As defined in Chapter IV, a project schema is an abstract view of the software development process. Annotated as a directed hypergraph, the nodes of the project schema represent different types of software development artifacts (components) while the hyperedges of the schema represent development activities (steps). Each of the nodes and edges is later decomposed into the instantiated atomic components and atomic steps of the development effort. The engineer creates the project schema by identifying all of the different types of software development artifacts created in their particular development process. Next the engineer identifies the activities that are used to produce these artifacts. Finally, the engineer connects these artifacts and activities into a process flow diagram that mirrors the software development process actually being used.

2. Establishing the Tool Ontology

Once the engineer has created the project schema, the next step is to apply the techniques described in Chapter III to create a foundational ontology that can be used to

unify the software development tools that produce the artifacts portrayed in the project schema.

3. Constructing the FIOM

The third step is to apply the techniques described in Chapter V to use the federation ontology to construct a FIOM of the software development domain. The individual tool ontologies provide the separate component representations used to construct the FIOM. The data from these tools (data representing the software development artifacts) is the data for which translators are established.

4. Establishing Communication Mechanisms

The fourth step in establishing the HFSE is to embed the translators created in the FIOM construction process into a communications mechanism so that data exchange and joint task execution is achieved. Middleware (such as CORBA) is one such candidate communications mechanism and potentially provides dynamic flexibility within the HFSE. However, the communications mechanism need not be as complex as a dynamic middleware implementation. For instance, the mechanism used in this dissertation to demonstrate theoretical feasibility of the HFSE was to use a simple static file import mechanism. The complexity of the communications mechanism should be selected on the basis of the complexity of the software development being modeled. A development effort involving numerous concurrent activities, customers, developers, architects, and programmers may require a more complex communications mechanism than one implementing a light, agile software development process.

5. Identifying Dependencies and Artifact Correlations

The final step in establishing the HFSE using the techniques described in Chapter IV is to create QFD dependencies, artifact (component) correlations, and to deploy the dependencies throughout the software development effort.

B. EXTENSIONS

1. Extending the FIOM with Additional Software Development Tools

Once a FIOM has been created that integrates a selected set of software development tools, the FIOM can be extended to incorporate new tools and remove obsolete tools. The techniques for adding tool representations are discussed in Chapter V

and in greater detail in [YOUN02b]. The basic concept is that the new Component Class Representations (CCRs) of the new tool are integrated into the existing Object Model of the tool Federation. The Federation Entities (FEs) are modified to accommodate the additional representations. Additional translators are constructed as needed.

2. Extending the HFSE

Extending the HFSE consists of two main steps: modifying the project schema and then integrating the updated FIOM into the framework. The HFSE project schema is simply redrawn (or modified) to accommodate the additional artifacts or the removal of artifacts. Next the undated FIOM is used to register the

3. Adding Dependencies

Adding additional dependencies to the HFSE is more straightforward than adding additional tools to the FIOM. New dependencies are simply created at any point of the development effort. Initial values for the dependency are established and the dependency values are deployed both upstream and downstream in the development effort.

C. FOCUSED DEVELOPMENT USING "SLICES"

1. Dependency Threshold

The HFSE allows the engineer to isolate particular subsets of software development effort as a smaller subgraph of the overall hypergraph that represents the entire development effort. Once such “slice” is induced by the Dependency Threshold View. For a particular dependency, the Dependency Threshold View is useful if the engineer wants to identify which components are associated with the greatest (or least) dependency values, for example the greatest risk, the least difficulty, etc. In each of these cases, the engineer would have defined that particular dependency (risk and difficulty) and assigned values to the dependency and deployed the dependency throughout the development effort. The Dependency Threshold View then provides a means of isolating important information from the remaining noisy data in the development effort.

2. Component Tracing

The second way in which the HFSE allows the engineer to isolate interesting subgraphs of the development hypergraph is through the Component Trace View. The Component Trace View is useful because it allows an engineer to identify all connected

components of a particular component of interest. For example, if the engineer wanted to modify a particular program module, he could use the Component Trace View to identify all connected components that influenced (or are influenced by) that particular module. Furthermore, because the correlations between components in QFD matrices all have particular values, the engineer can specify the degree of relationship he is interested in isolating.

3. Potential Application of Risk-Induced Slices

One possible future research thread is to examine the benefits of using the HFSE to extract a risk-induced slice of the development effort and to then focus developmental resources on that slice. This potentially would improve the Software Development Life Cycle in the following three ways:

a. Greatest Risk Slices

First, the HFSE aims to improve prototyping efforts by allowing the designer to focus only on those aspects of the design that represent the greatest uncertainty and/or greatest risk to project success. It should be possible to extract a "slice" of the entire dependency graph so that effort is not wasted on aspects which are already well defined, understood, and/or successfully implemented in previous versions. This will make the development effort more efficient and economic. Also, it may be possible to make this become a prescription rule rather than a filtering constraint, so that only the parts of the graph in (or near) the designed slice are constricted in the system to begin with. Building the entire graph and then "slicing" it may waste a lot of human effort.

b. Change Knock-on Effects

Secondly, it will be possible to identify and isolate the impact of individual changes on other dependent parts of the software development effort. Because all the software artifacts are linked together by a relational hypergraph, after making a single change to a single artifact it will be possible to have immediate visibility of all other artifacts that require modification due to that change. It will also be possible to have immediate visibility of the customer's view of the change since their priorities have been "deployed" (by SQFD) throughout the design. A possible implication of this is to

support cost-benefit analysis to determine if the proposed modification be with the effort -- or whether the effort can tolerate “the implied cost” of the modification.

c. Safety Certification

Finally, we can leverage the extended Evolution Model by identifying the total effect of individual changes to safety critical portions of the software. Producing reliable software is an expensive and time-consuming endeavor. Much of the expense and time is related to identifying all "knock-on" effects when modifying critical parts of the software and then re-certifying the software after accessing these knock-ons. The extended Software Evolution model will make it possible to identify all of these "knock-on" effects quicker and more meticulously by relying on automated methods.

D. CHAPTER SUMMARY

This chapter provided a high level summary of the Holistic Framework for Software Engineering. Specifically, it explained how to apply the HFSE to integrate a set of software development tools, how to extend the HFSE to include additional tools, and how to use the HFSE to obtain user defined holistic views of the software development process

THIS PAGE INTENTIONALLY LEFT BLANK

VII. EXTENSIONS TO THE COMPUTER AIDED SOFTWARE EVOLUTION SYSTEM (CASES)

A. CHAPTER OVERVIEW

As presented in Chapter II, the Computer Aided Software Evolution System (CASES) was developed in support of Harn's RH model of software evolution [HARN99c]. Specifically, CASES was developed by [LEHC99] to demonstrate the feasibility of providing tool support for the RH model. In order to support the work in this dissertation, the CASES tool was further extended to include needed functionality for the HFSE and the QFD extensions presented in Chapters IV and VI. As part of this dissertation, a set of UML use cases was developed that clarify how CASES is to be used and how it interacts with other tools in support of the HFSE (see Appendix A). The effort to extend CASES with QFD functionality was specifically undertaken by Clomera [CLOM03] and this chapter presents and summarizes the key contributions of his work in developing a software evolution support tool to support the construction of the HFSE. Readers interested in additional detail of the CASES extensions developed in support of this dissertation are referred to [CLOM03] and Appendix A.

This chapter provides an overview of the extensions made to CASES version 1.1 to create CASES version 2.0. Section B summarizes the results of the effort to provide graphic functionality for software development project schema creation. Section C presents the details of the QFD functionality embedded in the tool so that artifact dependencies can be created, tracked, and deployed. Finally, Section D summarizes the work that provides user defined views so that dependency data can be isolated, examined and reasoned about. The major contributions provided by [CLOM03] in support of this dissertation allow a software engineer to (1) design a custom software evolution model through the use of the CASES GUI, (2) input, modify, and analyze dependency characteristics between software artifacts within a QFD framework, and (3) make decisions based upon views of dependency information.

B. GRAPHICALLY DEFINING A SOFTWARE DEVELOPMENT PROJECT SCHEMA

1. The Project Schema in CASES version 1.1

In her Masters Thesis [LEHC99], Le developed the automated software evolution tool known as the Computer-Aided Software Evolution System (CASES version 1.1) in support of Harn's Relational Hypergraph (RH) Model of software evolution. CASESv1.1 was developed using object-oriented tools, Java Development Kit (JDK) 1.1.7, Swing 1.0.3, under the Visual Café version 3.0 environment (see Figure 57). CASES assists the software engineer in performing software evolution activities and allows the engineer to better control and manage the software evolution process. The tool provides five functions related to the activities of software evolution. These are step refinement, project evaluation, constraint management, personnel management, and step management. Additionally, CASES provides five functions related to software evolution components: component management, component traceability, configuration management, dependency management, and inference rule management.

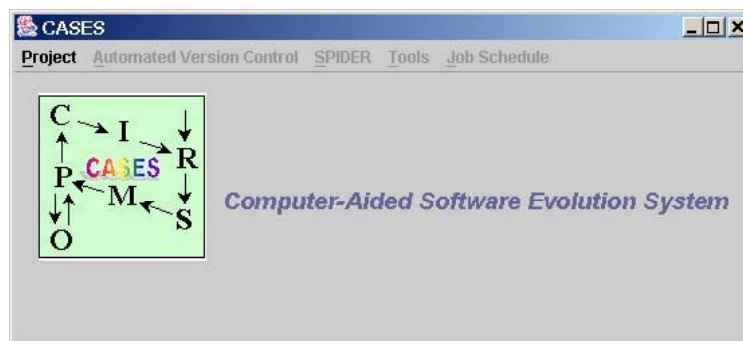


Figure 57 CASESv1.1 New Project Screenshot

CASESv1.1 allows the user to define steps and components tailored to a specific single software development methodology: the evolutionary process model (previously illustrated in Chapter II, Figure 6). CASES manages and controls all of the activities that change a software system and the relationships among these steps and components. CASESv1.1 was the first tool to support the practicality of the RH model. However, the utility of the tool was constrained by the reliance on a text and menu driven functionality tied to a single specific software development model. In version 1.1 of CASES, the

software engineer inputs information related to their software development process through a series of text and menu driven dialogs (see Figure 58).

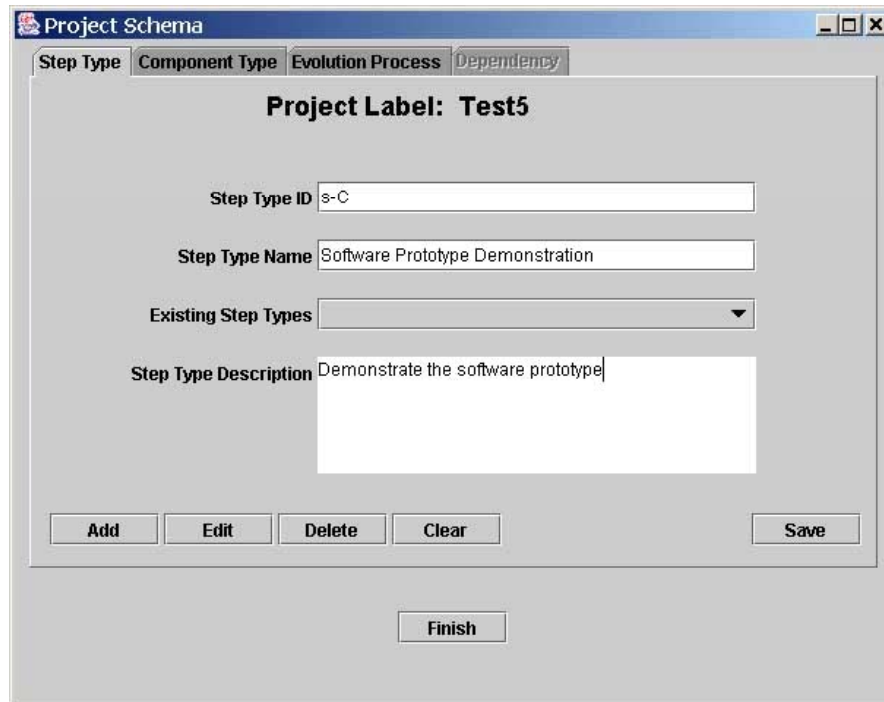


Figure 58 CASESv1.1 Project Schema Creation Dialog

While not only laborious, this approach obfuscates the evolutionary linkages within lengthy textual strings. One of the objectives in the research conducted by [CLOM03] was to improve Le's work by providing an intuitive, graphically based interface so that the software engineer could easily define project schemas related to their own specific software development process.

2. The Project Schema in CASES version 2.0

In CASES version 2.0, [CLOM03] improves upon version 1.1 by providing an interactive GUI by which software engineers can construct and tailor a specific software development project schema. The schema is no longer necessarily tied to the evolutionary prototyping model of software development. The engineer is free to create components and steps that represent the artifacts and activities actually used in the development effort, artifacts actually created in other software development tools. Figure 59 illustrates the CASESv2.0 project schema drawing pane in which an engineer is graphically depicting an abstraction of their particular software development process by

identifying unique components (types of software artifacts) and connecting those components with development steps (types of activities leading to the generation of components).

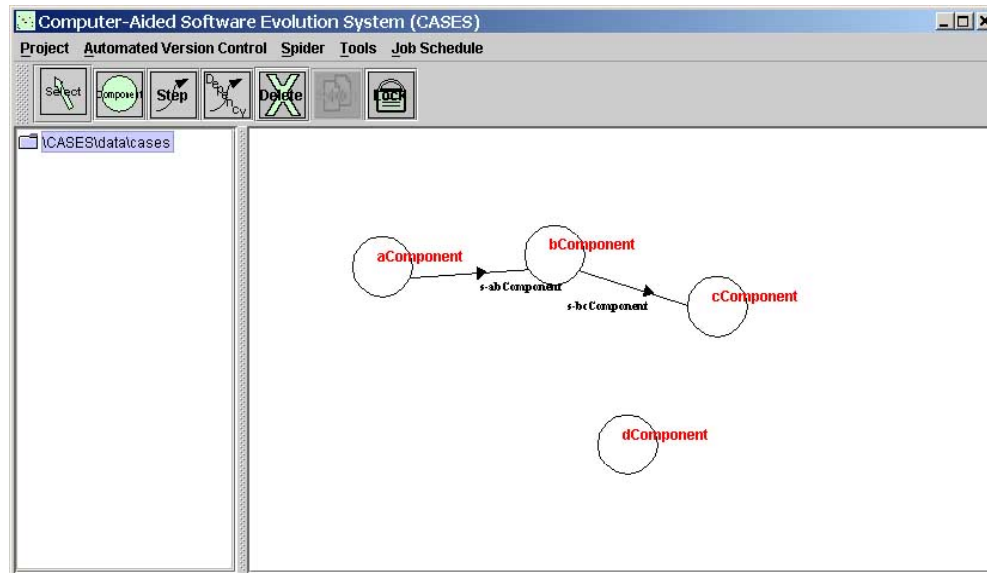


Figure 59 CASESv2 Project Schema Creation Process

In this particular case the engineer uses a point-and-click interface to create the project schema. As the engineer creates new components and steps, the components and steps are automatically given a unique identifier name. The individual components and steps are then specialized based on the specific attributes of the development process to complete the project schema.

As an example, consider a simple software development life-cycle in which the main development artifacts include the following:

- Customer requirements,
- Questions and answers (based on customer-developer dialog about the requirements),
- Software specifications,
- Code, and
- Feedback (from the customer based on validation testing of the code).

The project schema for this particular software process is illustrated in Figure 60.

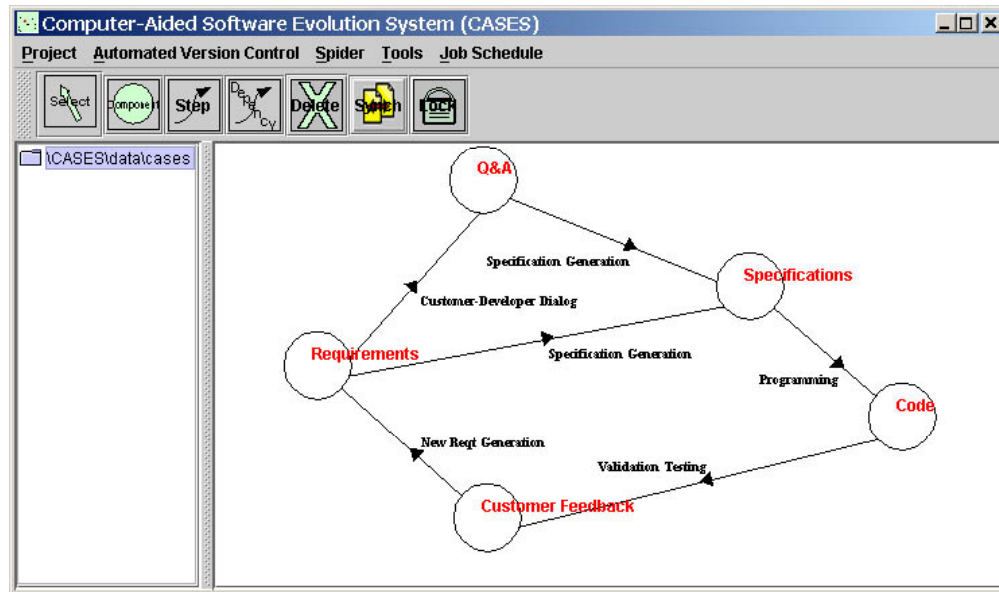


Figure 60 CASESv2 Completed Project Schema

Note that in this schema, the different types of components (the different types of software development artifacts) are abstracted as individual vertices in a directed graph and the steps (the activities in the development effort) are abstracted as directed edges in the graph. This easy to understand abstraction simplifies the textual based project schema format used in CASESv1.1. Notice also that this schema bears no resemblance to the evolutionary prototyping model required in version 1.1, the engineer was able to create a project schema tailored to and matching their own unique software development process.

It is notable, that while the use of the evolutionary prototyping model is not required in CASESv2.0, there is no reason a software engineer could not create that particular schema if desired. The schema created in Figure 61 illustrates exactly that -- the IBIS evolutionary prototyping model (originally illustrated in Chapter II, Figure 6) implemented in CASESv2.

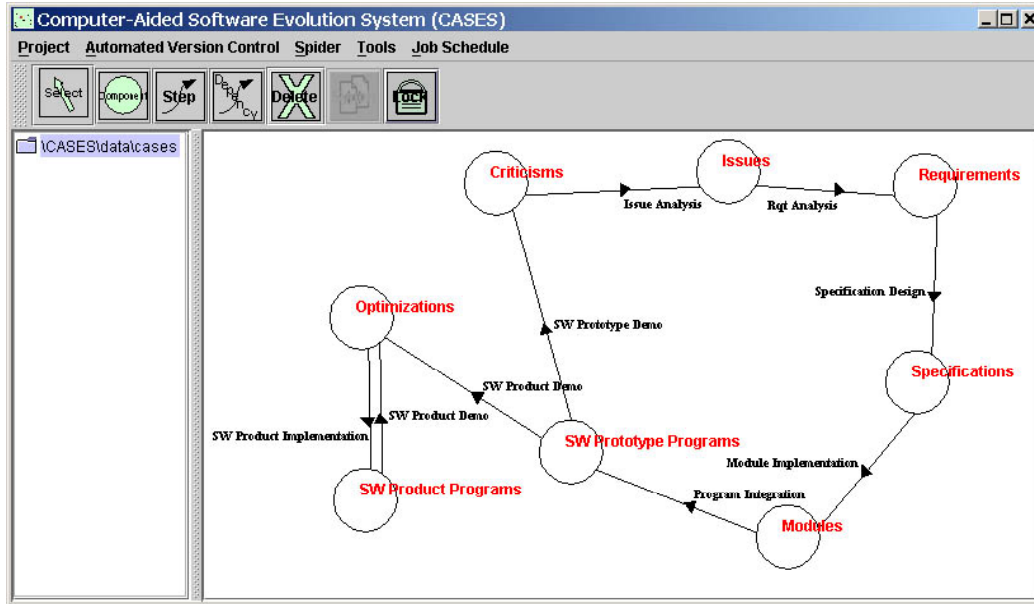


Figure 61 IBIS Evolutionary Process Model in CASESv2

Providing a project schema creation GUI is a significant enhancement over version 1.1. The GUI provides the software engineer an easy to use, intuitive interface for modeling within CASES the actual artifacts and activities used in particular software development efforts. Next, consider the CASES enhancements related to QFD that allow the engineer to decompose these high-level component and step abstractions into atomic components and steps so that the engineer can deploy particular defined dependencies throughout the whole of the development effort.

C. EMBEDDING QFD INTO CASES

There are three main enhancements needed to CASESv1.1 in order to embed the QFD functionality required to support the HFSE. First, CASES needed a means of capturing user specified dependencies (and attributes of those dependencies). Second, CASES needed a means of registering the abstract components of the project schema (e.g. requirements, specifications, code, etc.) with specific decomposed, instantiated component data (e.g. atomic components such as Requirement number 1.3, variant 2, version 3). Third, CASES needed to provide the engineer the ability to display and modify QFD matrices implied by the registered data.

1. QFD Dependencies

As defined in Chapter IV, QFD Dependencies are the set of specialized relationships that are “deployed” from one set of components to another. Examples of QFD Dependencies include risk, customer priority, difficulty of implementation, cost of implementation, requirement stability, safety, security, etc. In CASESv2.0 QFD Dependencies begin with the use of a Dependency creation dialog (see Figure 62). The engineer creates dependencies by left “clicking” the “dependency” button on the CASES toolbar. The engineer inputs the dependency name, description, type, value range, default value and origin.



The screenshot shows a 'Dependency Attribute' dialog box. It has a title bar with a close button. The fields are as follows:

Field	Value
Name	Reqd Risk
Description	The Customer's view of risk
Type	Risk
Value Range	0-9
Default Value	0
Origin	Requirements

Buttons: Save, Cancel

Figure 62 CASESv2 Dependency Creation Dialog

The name provides a short name for identifying one dependency from another. The description provides the engineer the opportunity to record particular details of a dependency so that others can later understand the context of the dependency. The type can be selected from “risk”, “safety”, and “parent-child.” In CASESv2.0 the “type” field does not provide any actual functionality; but is provided as a possible future extension in which specific, pre-defined dependency type attributes can be associated at run-time with particular instantiated dependencies. The value range is also provided as a feature for future extension and currently assumes a real value range of 0 to 9. Future extensions of value ranges could account for step functions, integer values, or Boolean values. The default value is used to initialize the dependency values in the QFD matrix upon the first

use of a dependency. The “origin” of the dependency must be set to a component in the project schema and represents the source from which the dependency value is generated. For instance, a dependency of “Specification Difficulty” might use the “Specifications” component as the origin since it is likely that the difficulty value will be generated by considering the difficulty of implementing each specification. This origin is used as the basis to perform upstream and downstream calculations (discussed in Chapter IV) based upon that component.

2. Component Data Import

The next major QFD extension to CASES is the ability to import real artifact data from external software development tools. In CASESv1.1 the user was provided the functionality of opening an external software development tool (e.g. Microsoft Word, CAPS, MS Excel, Netscape, Notepad). Those tools were then used to create separate individual files for each atomic component. For example in [HARN99c], an individual text file “/c4idata/1.2/requirements/c4i.gui_2.5.req.text.txt” was created for this variant 1, version 2, requirement 2.5, C4I GUI requirement:

The dynamic output graphic interface must provide the function of monitoring the target and missile intersection point.

This individual text file was then associated within CASES as a single reference to the atomic component: *R1.2-2.5*. While working within CASES, the engineer either had to remember what this reference referred to or had to create a separate listing of all the files and individual data in order to laboriously identify the linkages between components. As an example, consider Figure 63 and the effort required to obtain an intuitive feel for how *R1.2-2.5* was arrived at during the requirement analysis step. The engineer would probably have to have a minimum of four separate files open at the same time (related to the references *R1.2-2.5*, *R1.1-1.6*, *II.2-3*, *VT-R1.2-2.5*).

Requirement analysis step: *s-R1.2*

$(R1.2-1 \leftarrow s-R1.2-1 (R1.1, II.2-2, II.2-3, VT-R1.2-1))$
 $(R1.2-1.1 \leftarrow s-R1.2-1.1 (R1.1-1, II.2-2.1, II.2-2.2, VT-R1.2-1.1))$
 $(R1.2-1.2 \leftarrow s-R1.2-1.2 (R1.1-1, II.2-2.3, II.2-3.1, VT-R1.2-1.2))$
 $(R1.2-2 \leftarrow s-R1.2-2 (R1.1-1, II.2, VT-R1.2-2))$
 $(R1.2-2.1 \leftarrow s-R1.2-2.1 (R1.1-1.1, II.2-3, VT-R1.2-2.1))$
 $(R1.2-2.2 \leftarrow s-R1.2-2.2 (R1.1-1.2, II.2-3, VT-R1.2-2.2))$
 $(R1.2-2.3 \leftarrow s-R1.2-2.3 (R1.1-1.4, II.2-3, VT-R1.2-2.3))$
 $(R1.2-2.4 \leftarrow s-R1.2-2.4 (R1.1-1.5, II.2-3, VT-R1.2-2.4))$
 $(R1.2-2.5 \leftarrow s-R1.2-2.5 (R1.1-1.6, II.2-3, VT-R1.2-2.5))$
 $(R1.2-2.6 \leftarrow s-R1.2-2.6 (R1.1-1.2, II.2-3, VT-R1.2-2.6))$
 $(R1.2-2.7 \leftarrow s-R1.2-2.7 (R1.1-1.6, VT-R1.2-2.7))$
 $(R1.2-2.8 \leftarrow s-R1.2-2.8 (R1.1-1.2, VT-R1.2-2.8))$

Figure 63 C4I Systems Requirements Analysis Step (after [HARN99c])

Thus, there are a number of shortcomings with this functionality. First, each file association had to be individually created and managed, a laborious process. Second, because actual artifact data was not used in CASES (only references to the data), it was difficult for the engineer to quickly glean meaningful information from the references. CASESv2.0 addresses these shortcomings by providing an “import” function in which the user is able to import atomic component data directly into the tool. This situation is improved in CASESv2.0 where the engineer gains an intuitive feel for the relationships between atomic components through use of the project schema and individual QFD matrices.

In CASESv2.0, the software engineer registers the high level abstract component types in the project schema with specific atomic component data contained in .csv (comma separated value) files produced from individual tools. CSV files are static flat files easily created from ASCII text, spreadsheets, or database files. The engineer “imports” this atomic component data directly into CASES by “right-clicking” on a

component, identifying the particular variant and version number and then navigating to the .csv file (see Figure 64).

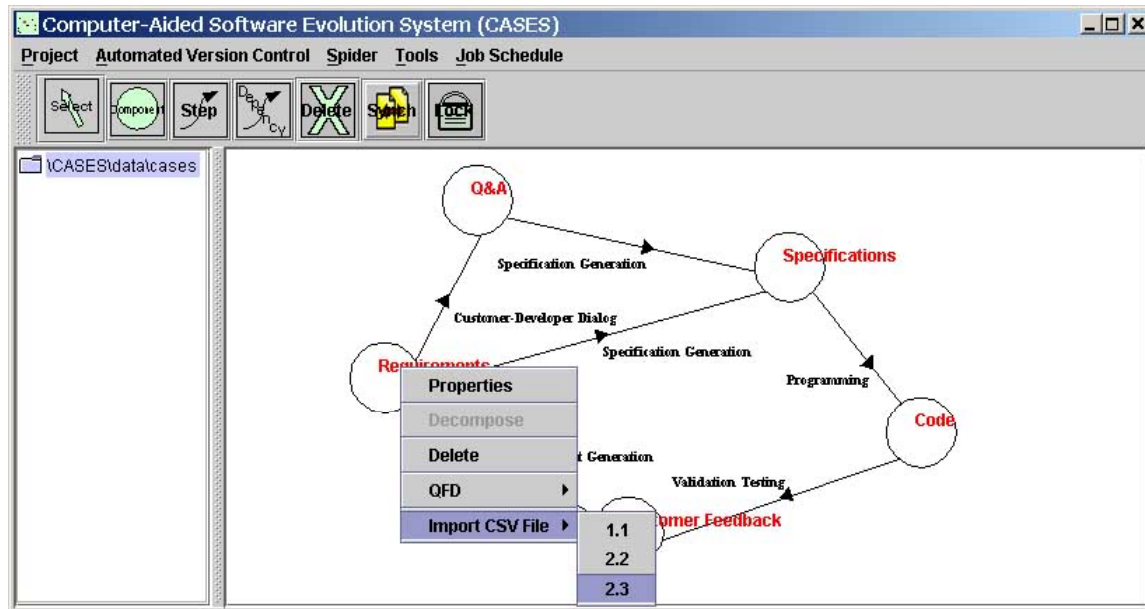


Figure 64 CASESv2 Data Integration via Import CSV File (Requirements variant 2 version 3)

In Figure 64 the engineer imports the .csv file for the Requirements of variant 2 version 3. This data can then be seen and used in establishing QFD relationships within the CASESv2.0 QFD Matrices.

3. QFD Matrices

The third major extension needed to embed QFD into CASES is to provide a QFD matrix capability so that the engineer can establish correlations between components, enter or modify dependency values, and then perform upstream and downstream deployment calculations. The engineer views and modifies a particular QFD matrix by “right-clicking” on a particular step (edge) and selecting “QFD” followed by the particular dependency of interest. The user is then presented with a QFD matrix with the input component on the left and the output component across the top (see Figure 65). The engineer is then free to enter correlation values (typically 0, 1, 3, or 9 (see Chapter IV)) and to enter or modify dependency values. Dependency values may only be edited at the “origin” component (specified during dependency creation).

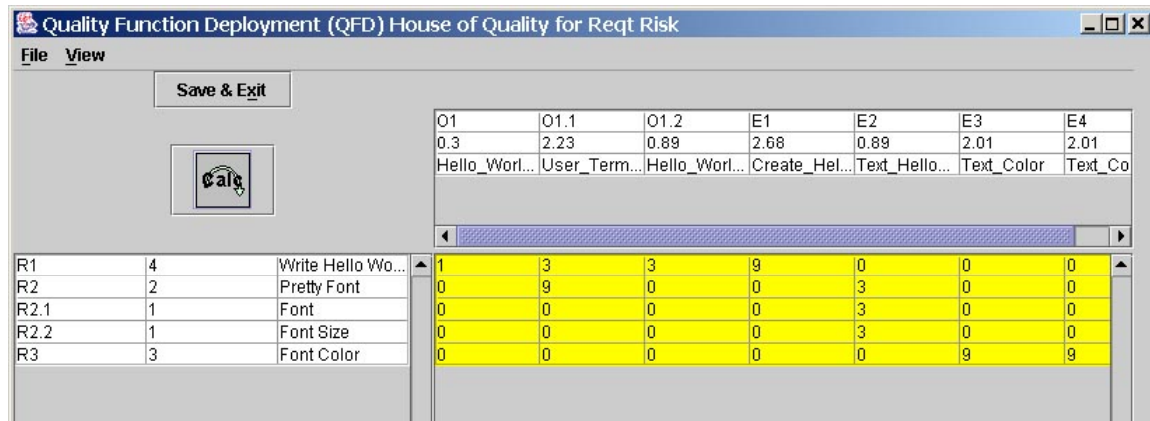


Figure 65 QFD Matrix: Requirements x Specifications (Dependency: Reqt Risk)

Finally, after all the correlation dependency values are entered, the user presses the “Calc” button the on the QFD dialog or the “Sync” button on the CASES toolbar and the tool automatically “deploys” the dependency. In the case of the “Calc” button, the dependency is only deployed to the components of the open QFD matrix. In the case of the “Sync” button, the dependency is deployed to all connected components in the software development effort.

In Figure 65 the engineer was presented with a QFD matrix between “Requirements” and “Specifications” for the QFD dependency “Requirement Risk”. Note that the imported textual information (the requirement/specification numbers and names) was automatically displayed and allows the engineer to intuitively understand what linkages and comparisons are being established. The engineer could have imported the values for “Requirement Risk” via the .csv file or entered/edited those values individually. After entering in values of correlation between the requirements and the specifications, the engineer used the “Calc” button to deploy the Requirement Risk to the specifications.

D. ENGINEERING VIEWS OF QFD DEPENDENCIES (SLICES OF THE RH MODEL)

Simply creating QFD matrices and entering information into them does little to assist software engineers in building better software or in improving their software processes. In order to offset the overhead associated with gathering and entering this information, there must be means by which the engineers can quickly isolate and view

interesting information that will help them improve their software product or improve their software development process. CASESv2.0 implements two such means by allowing the user to define two different views of QFD information: the Dependency Threshold View and the Component Trace View.

1. Dependency Threshold View

The Dependency Threshold View allows the engineer to isolate a subset of important (or unimportant) components from the entirety of the development effort. This view is useful if the engineer wants to identify which components have the greatest risk, or are the least difficult, or are the most unstable. In each of these cases, the engineer would have defined that particular dependency (risk, difficulty, stability), assigned values to the dependency and deployed the dependency throughout the development effort. The Dependency Threshold View then provides a means of isolating important information from the remaining noisy data in the development effort.

To use the Dependency Threshold View the engineer completes all correlation matrices and uses the synchronization function on the CASES tool bar to deploy all dependencies throughout the entire development effort. Next, the engineer opens a particular QFD matrix of interest. From the “View” menu item of the QFD dialog, the user selects “Dependency Threshold” and is presented with a dialog in which he can select a threshold of interest based on the mean and standard deviations of the components displayed. The mean value +/- the standard deviation of the dependency values are used as opposed to specifying a particular threshold value because of the dependency “thinning” and “concentration” effects discussed in Chapter IV.

To illustrate the Dependency Threshold View in CASESv2.0, recall the example presented in Chapter IV where three requirements were deployed to four specifications (see Figure 66).

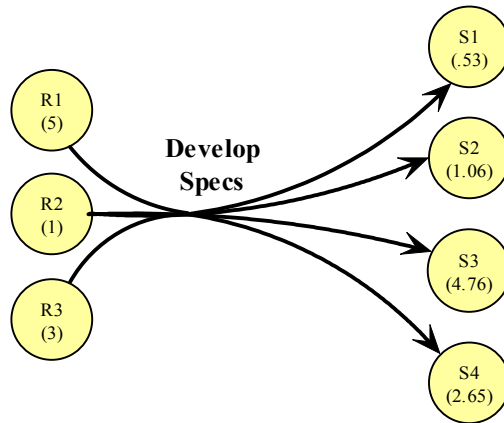


Figure 66 User-Defined Example from Chapter IV

The CASESv2.0 QFD correlation matrix for this example is presented in Figure 67. This matrix corresponds exactly with the data presented in Chapter IV, Table 12.

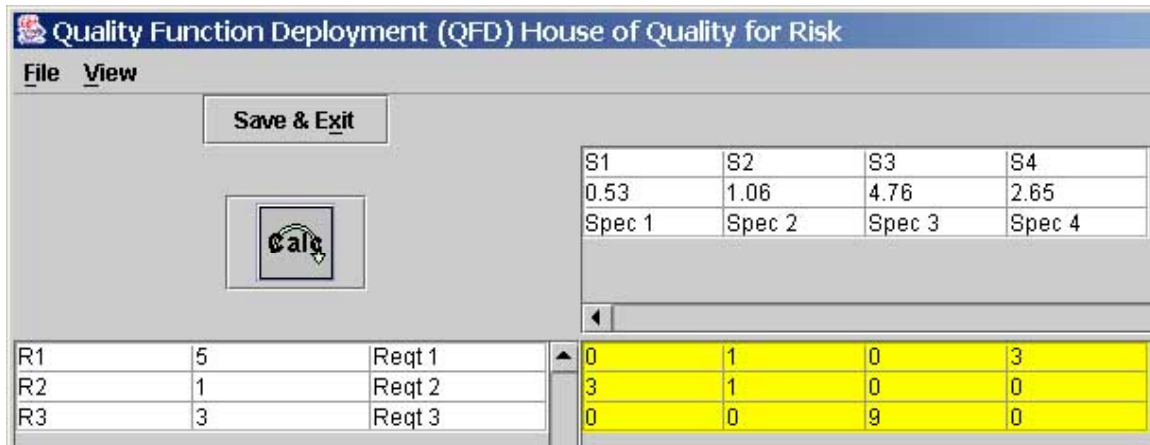


Figure 67 QFD Matrix: R x S (Dependency: Risk)

From this QFD matrix, the engineer selects the “View” menu item and the “Dependency Threshold” option. As in the example presented in Chapter IV, if the user desires to view all components greater than the mean, he enters that into the dialog and the tool automatically trims the QFD matrix to show only those components meeting that criteria. The user can specify additional thresholds and the tool will continue to provide the appropriately trimmed matrices. Figure 68 illustrates two such matrices consistent with the example in Chapter IV in which the user desired to view all components with dependency values greater than the mean (upper left matrix) and a separate matrix of all components with dependency values greater than one standard deviation above the mean

(lower right matrix). Note that the resulting components match exactly those derived analytically as in the example in Chapter IV.

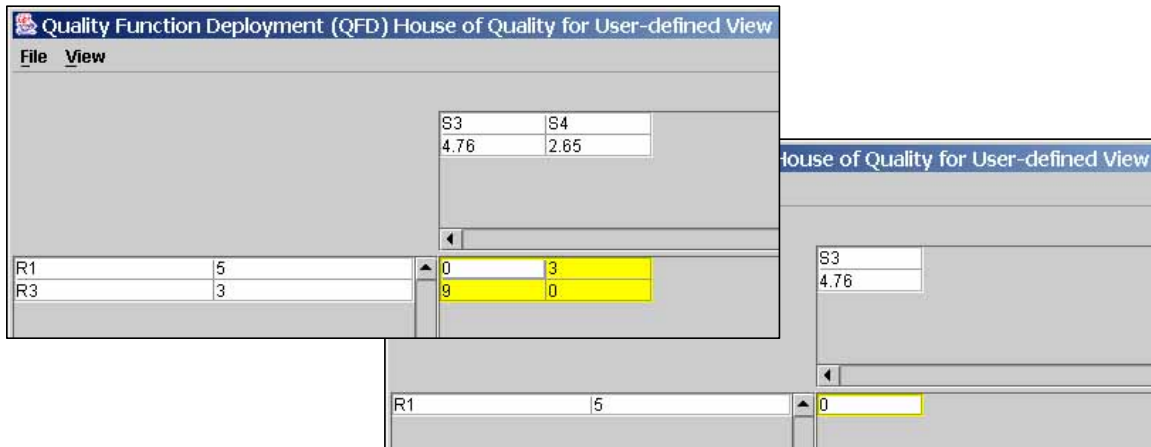


Figure 68 User-Defined Views with Threshold = μ and Threshold = $\mu + 1\sigma$

It is worth noting that these trimmed matrices equate to induced subgraphs of the underlying hypergraph representation of the entire development effort. These subgraphs have been induced by applying user defined threshold criteria against the deployed QFD dependency values.

It is clear that by using such views the engineer could quickly identify specific subsets of development components that are of interest. For example, the engineer could quickly identify all the components that have the greatest risk, or identify the components that are most related to safety requirements provided by the users. Conversely, the engineer could quickly identify which components have little risk or no safety implications. Such specialized informational views provide important decision support information allowing software engineers to make better decisions as to where to allocate limited development resources. The Dependency Threshold View could be of benefit in numerous development situations and is only constrained by the type of dependencies created by the engineer and the engineer's ability to accurately determine dependency values and component correlations.

2. Component Trace View

The Component Trace View is useful to the engineer so that he can identify all connected components of a particular component of interest. For example, if the engineer were having difficulty in implementing a particular architectural component, he

could use the Component Trace View to identify all connected components that influenced (or are influenced by) that particular component. Furthermore, because the correlations between components in QFD matrices all have particular values, the engineer can specify the degree of relationship he is interested in isolating.

To use the Component Trace View the engineer completes all correlation matrices and uses the synchronization function of CASES tool bar to deploy all dependencies throughout the entire development effort. Next, the engineer opens the QFD matrix just upstream of the component of interest (i.e. so that the component of interest is listed across the top of the QFD matrix). From the “View” menu item of the QFD dialog, the user selects “Component Trace” and is presented with a dialog in which he can select the particular component of interest and the threshold of correlation. For example if he selects a threshold of 2, CASES will display all connected components that have correlation values of 2 or greater.

To illustrate the Component Trace View in CASESv2.0, recall the example presented in Chapter IV with requirements, specifications, architectural components, and software modules connected by three steps (see Figure 69).

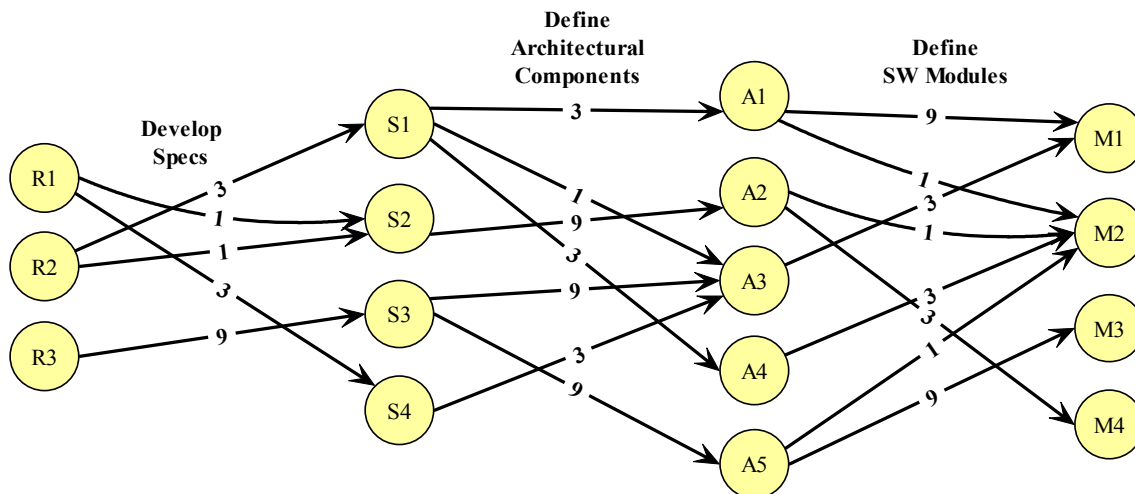


Figure 69 Component Trace Example from Chapter IV

The CASESv2.0 QFD correlation matrices for this example are presented in Figure 70. These matrices correspond exactly with the data presented in the component trace example from Chapter IV and in data in Figure 69.

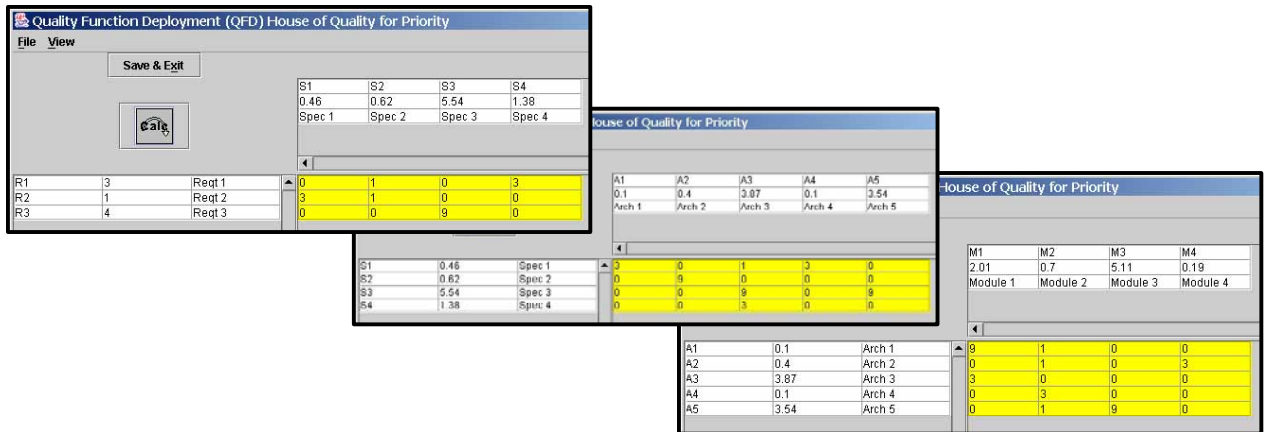


Figure 70 QFD Matrices for Component Trace Example

Now suppose the user wants to perform a component trace from component A3 (one of the architectural components). The engineer would select the middle matrix from those in Figure 70 (the matrix in which component “A3” is listed across the top). From this QFD matrix the user would select the “View” menu item and the “Component Trace” option. In the resulting dialog, the user would specify which component they desire to trace from and at what threshold value. CASESv2.0 then presents the user with a series of “trimmed” QFD matrices corresponding to each major step in the development effort in which the resulting trace exists. Figure 71 and Figure 72 illustrate the resulting trimmed QFD matrices related to component traces from component A3 with threshold values of 2 and 8 respectively.

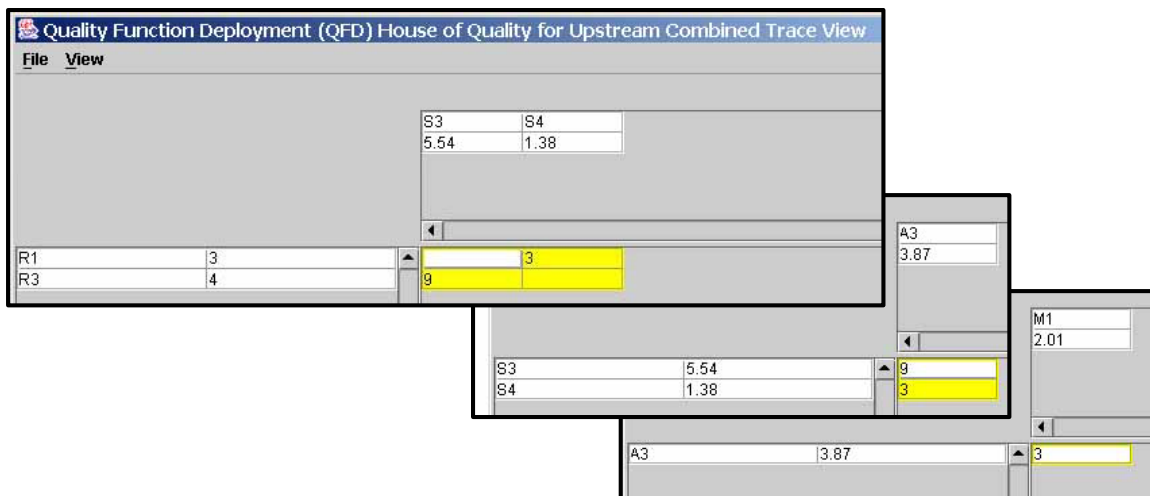


Figure 71 QFD Trace from A3 (Upstream & Downstream) Threshold 2

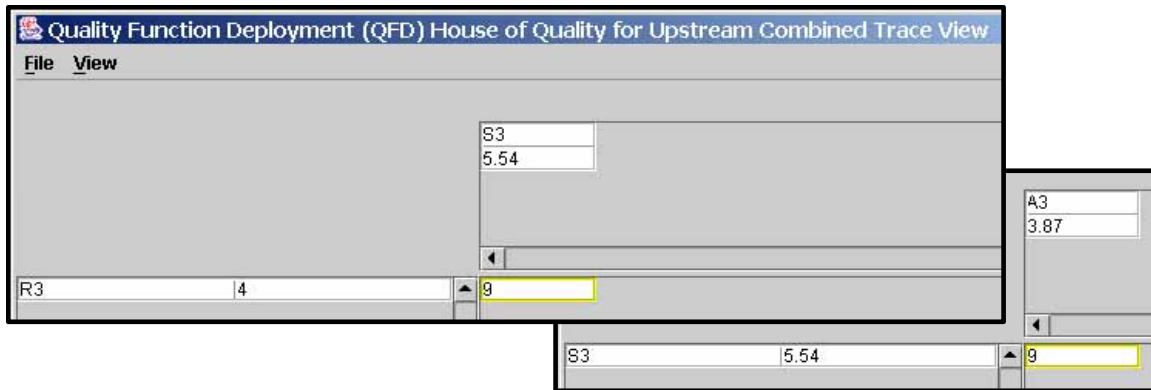


Figure 72 QFD Trace from A3 (Upstream) Threshold 8

As in the case of the Dependency Threshold Views, it is important to note that these trimmed matrices equate to induced sub-graphs of the underlying hypergraph representation of the entire development effort. These figures equate exactly to the sub-graphs derived analytically in the example in Chapter IV (recall Figure 45 and Figure 46). These sub-graphs have been induced by applying user defined threshold criteria against the QFD correlation values based on linkages established in the project schema.

Once again it is clear that by using such views the engineer could quickly identify specific subsets of development components that are of interest. For example, if the engineer were having difficulty implementing an important software module, the engineer could quickly view a trace of all previous components (architectural modules, software specifications, customer requirements) that led to the need for that software component. The engineer could then reason about ways in which the architecture could be modified or the requirement renegotiated in order to find a workable and agreeable solution to the implementation problem. Alternatively, suppose a change was being proposed to a safety critical requirement, the engineer could use the Component Trace View to identify the potential knock-on effect of the proposed change to quickly seeing all the effected components which would be impacted by the change in the single requirement component. As was the case with the Dependency Threshold View, the Component Trace View is a specialized informational view that provides important decision support information allowing software engineers to make better decisions as to where to allocate limited development resources.

E. CHAPTER SUMMARY

This chapter presented the results of the research effort devoted to providing tool support for the HFSE. It provides an overview of the extensions made to CASES version 1.1 to create CASES version 2.0 by [CLOM03]. The major contributions provided by this improved version of CASES allow a software engineer to: design a custom software evolution model through the use of the CASES GUI; input, modify, and analyze dependency characteristics between software artifacts within a QFD framework; and make decisions based upon views of dependency information.

Together, these extensions provide the necessary tool support that allow the establishment of the HFSE and allow the HFSE to be applied in actual software development scenarios.

VIII. VALIDATION

A. APPROACH TO VALIDATION

Recall that the dissertation hypothesis (presented in Chapter I) called for determining the theoretical feasibility of using the HFSE to improve the interoperability of software development tools and models. The overall approach to providing evidence confirming this hypothesis was to apply the HFSE to two software development tools and then use those tools in two software development scenarios. However, before discussing the experiment, the results, and implications of the results, the research hypothesis itself begs the following questions that must be addressed:

- What is meant by “theoretical feasibility”?
- What is meant by “interoperability”?
- What is meant by “improvement” (of interoperability)?

This section of the chapter will define these ambiguous terms, will present the experimental approach, and will discuss the employed risk mitigation and experimental scoping measures.

1. Definitions

a. Theoretical Feasibility

Generally, there are three categories of feasibility to consider when conducting scientific investigation: theoretical, technical, and organizational. In theoretical feasibility the underlying concern is demonstrating that something could be accomplished (the experimental variable operating on the observation group producing an effect on the observation group) if adequate resources existed and were then applied to the effort. Theoretical feasibility can be demonstrated through mathematical proof or by constructing a proof of concept technical prototype or simulation. In technical feasibility, the underlying concern is demonstrating that all needed resources exist and are available for completely accomplishing the investigative task. Technical feasibility is demonstrated by completely implementing the process under investigation using established technology and then demonstrating that it works in real world situations. In organizational feasibility, the underlying concern is demonstrating that while something

might be possible and the technology exists to support it, that there are organizational reasons why it should be done.

In the case of this dissertation, only evidence of theoretical feasibility is established. There are two main reasons for this. First, the effort needed to demonstrate technical feasibility of the HFSE by actually integrating all existing software development tools via a single ideal dynamic middleware solution is a massive undertaking requiring many hundreds (if not thousands) of man-years of effort. Second, the ideal middleware mechanism used to implement the HFSE is not yet identified (and is considered as an area for future research). In this dissertation, the manual exchange of static flat data files is used to demonstrate the theoretical feasibility of the framework and provides evidence that one such technical solution is possible, but by no means should this simplistic middleware solution be viewed as the ideal technical implementation. In a technical feasibility study, several dynamic middleware solutions should be compared in order to determine the most efficient one for implementation within the HFSE for use in all real world settings. Finally, organizational feasibility is not considered at all in this dissertation. While an organization's rationale for potentially employing the HFSE is anecdotally discussed, no experimental evidence is provided that would demonstrate conclusively that real organizations have a desire (or an unaccounted for economic impetus) to implement the framework.

b. Interoperability

There are many definitions of interoperability. For instance, NATO definitions of interoperability focus on data exchange and define categories (levels) of interoperability based on the type of data exchange possible between two systems [STAN00]. Level 1 interoperability consists of separate operators at two different types of systems manually entering in similar data into their systems; level 2 interoperability consists of the exchange of media (floppy disc, CDROM, etc.) which can be read by two different systems; level 3 is the electronic exchange of character based message sets which can be interpreted by different types of systems; etc. In all cases, the NATO definition of interoperability focuses on the means, ease, speed, and capacity of *data exchange*.

[GANG00] provides an alternative view of interoperability based on four perspectives: physical interoperability, data-type interoperability, specification-level interoperability, and semantic interoperability. Physical interoperability relies on the physical exchange of compatible electronic storage media. Data-type Interoperability focuses on content and structure of the information exchanged. In specification-level interoperability, applications that share data do not know the finer details of the data structure, but rather treat information to be shared as a whole; COM and CORBA are examples. In semantic interoperability, a system is designed to use different abstract views of shared entities.

The definition of interoperability that will be used in this dissertation is the same one that [YOUN02b] relied on for his OOMI methodology and is attributed to [PITO97]:

Pitoura defines interoperability as the capability of systems to exchange information and to jointly execute tasks. Full interoperability allows systems to take advantage of functionalities and services that would otherwise not be available or would have to be implemented.

In this definition, the concept that interoperability is more than just data exchange is key. The ability of users of the systems to accomplish joint tasks beyond the individual capabilities of each system is as an important aspect of interoperability as is information exchange. It will be this “joint task execution” portion of interoperability (using artifact information from different systems to identify dependency relationships) that will be highlighted during the dissertation experiment.

c. Interoperability Improvement

In this dissertation “improvement” in interoperability will be considered as any increase in quantity, type, or speed of information exchange as well as any increase in quantity, type, or speed of joint task execution. For instance, if after the application of the HFSE to a set of software development tools, the tools can execute new joint tasks (tasks they were not able to undertake prior to the application of the HFSE) this will be considered an “improvement” to interoperability between the tools. Conversely, if two tools integrated via the HFSE can no longer exchange information or execute some

particular joint task that was possible before application of the HFSE, this will be considered to be a “decline” or “reduction” in interoperability.

2. Experimental Design

a. Overview

The experimental design of this dissertation involves applying the HFSE to a small subset of tools and then using that subset of tools in two software scenarios. A static group comparison will be undertaken to identify if “improvements” to “interoperability” exist in the integrated tool set when compared to the interoperability available between the tools in the same development scenario without benefit of integration by the HFSE. In essence, the experiment relies on a small representative subset of tools/models to show that the HFSE can be used to unify them and to provide evidence that the interoperability of the subset of tools is improved. In the discussion of internal and external experimental validity, theoretical arguments are then provided to characterize the class of tools and models that could also be unified with additional effort.

b. Static Group Comparison

This dissertation relies on a static group comparison test to provide confirming evidence of the dissertation hypothesis. Campbell and Stanley [CAMP63] point out that this comparison is best characterized as a "pre"-experiment because it falls short of the unbiased application of the scientific method (the sources of invalidity of the experiment are delineated later in this chapter). This experiment can be characterized as an application of the experimental variable (X) upon an observation group (O) as shown below in Experiment 1 (the experimental notation is adopted from [CAMP63] in which the horizontal line represents the experimental comparison of observation groups).

$$\frac{X \quad O}{O}$$

Experiment 1

In this experiment

$O \subseteq \{\text{all software development tools and models}\}$, and
 $X \equiv \text{Application of the HFSE to } O$.

A static group comparison experiment "is a design in which a group which has experienced X is compared with one that has not, for the purpose of establishing the

effect of X" [CAMP63]. In this specific dissertation experiment, the HFSE is applied to a selected subset of tools/models (Rational's Requisite®Pro and SEATools). The performance of integrated subset of tools/models (after the application of the HFSE) is then compared to the performance of same tools/models in a "stand alone" mode (i.e. without the benefit of integration by the HFSE). The two groups are separately applied to the same software development scenarios. The comparison in this case will be to determine if there are any improvements in interoperability between the tools (i.e. improvements to data exchange and/or joint task execution). Specifically, the experiment will seek to accumulate evidence of additional data exchange and additional joint task execution enabled by the application of the HFSE to the subset of tools/models. The experiment will also record counter-evidence that the HFSE reduces (or inhibits) data exchange and/or joint task execution.

The two software development scenarios consist of a small "toy" example involving several development iterations and a more complex real world software development effort focused on just a single portion of a development effort. The toy example consists of developing the classic "Hello World" application. While in the toy example the number of requirements, specifications, code modules, etc. are each relatively few (1 to 5 atomic components each), the scenario is illustrative of how the HFSE helps to holistically integrate many different types of artifacts with several variants and versions each. The real world scenario consists of the development of five parallel software variants of the CARA infusion pump from the same set of software requirements. This scenario consists of just a few different types of components (requirements, questions & answers, and specifications) but with numerous atomic components in each (e.g. over 150 requirements).

3. Sizing the Dissertation Investigation and Risk Management

Because of the potential size of this dissertation research and the complexity of existing models and tools that needed to be analyzed, the following mitigation measures were employed:

a. Limited Number of Software Development Tools Analyzed

Only two software development tools were analyzed to any significant depth. The original dissertation plan of research called for the analysis of five

models/tools [PUET02a, b] in establishing the software development tool ontology. However, investigating two tool ontologies (see Chapter III) provided sufficient analysis for proving the theoretical feasibility of the HFSE. The remaining effort was modified appropriately.

b. Superficial Exploration for Counter-examples

Once the analysis of the models and tools was complete and a characterization for them was established, an exploration for counter-examples was undertaken with the aim of improving the model. This consisted of the top-down approach used in developing the software development tool ontology (see Chapter III). This exploration was superficial in both breadth and depth, which is why the ontology was constructed to be extensible, so that additional future additions and modifications can be easily implemented in future research.

c. Partial Implementation of Example Tools

The primary goal of this research was only to formulate a framework that could be used to unify any/all software development tools and models, not one that did unify them all. This research did not seek to implement a complete federation of models and tools, nor did it seek to implement completely the tools considered (Requisite®Pro and SEATools). For instance, when incorporating Rational's Requisite®Pro into the HFSE, only limited functionality was incorporated and used, not the entire functionality offered by the tool.

d. No Validation against a real-world system

The HFSE was not validated in a complete real-world software development effort. Both the “toy” scenario and the “CARA” scenario were accomplished as incomplete laboratory development efforts. While the software requirements for the “CARA” scenario are from a real-world software development scenario, the remainder of the CARA development artifacts (e.g. the software specifications) stem from a research environment.

e. Early Implementation

While still investigating parts of the HFSE, the extensions to CASESv1.1 (leading to CASESv2.0) were ongoing. Setting up the HFSE sooner rather than later helped to bring to light new ideas and concepts that were incorporated into the effort.

Handling the most simplistic pieces of software in the HFSE allowed examination of the representation, communication, and temporal considerations of evolution of the development effort and identified needed additions and modifications to the extensions incorporated into CASES.

B. CONDUCT OF THE EXPERIMENT

In the conduct of the dissertation experiment, the HFSE was applied to Rational's Requisite®Pro and SEATools. The framework was then used in two software development scenarios: first, in a "toy" scenario consisting of a illustrative yet small development effort encompassing several complete development cycles with multiple versions and variants, and secondly, in a complex real-world example involving only the interaction between requirements and a single variant of software specifications. Each example provides confirming evidence of the dissertation hypothesis.

1. "Hello World" Toy Software Scenario

The first software development scenario conducted in the dissertation experiment is the classic "toy" example of "Hello World" which helps to clarify how the HFSE can be applied to software development efforts consisting of multiple variants and versions. In this scenario a customer and developer interact to produce multiple variants and versions of a software application that displays "Hello World" on a computer screen. Together, the customer and developer apply a software development process shown in Figure 73.

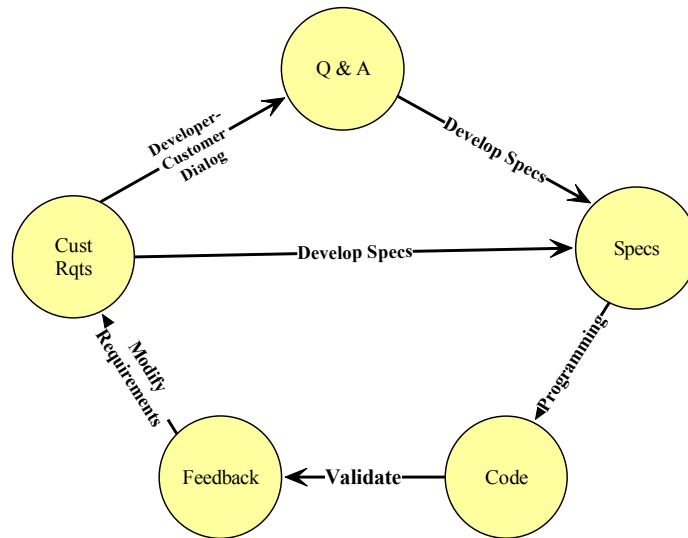


Figure 73 Hello World Development Process

The customer begins by stating his requirements in Requisite®Pro (see Table 29). The customer identifies the priorities of his requirements using both the requirements priority attribute inherent in the Requisite®Pro tool as well as creating a new requirements attribute called “AHP Priority” and then using the Analytic Hierarchy Process (AHP) [SAAT80] to provide values for that priority. The customer also creates a second user defined requirements attribute called “Risk.” He assigns to risk, values associated with the potential (probably of outcome and level of consequence) for economic loss, should that particular requirement not be implemented as planned (1=low risk, 5=high risk).

R Tag	Requirement text	Name	ReqPro Priority	AHP Priority	Risk
R1	When commanded, the software must display the text "Hello World" at the terminal.	Write Hello World	High	0.75	4
R2	The text will be pleasing to the user.	Pretty Font	Medium	0.0	2
R2.1	The font will be Arial.	Font	Low	0.05	1
R2.2	The font size will be 12.	Font Size	Low	0.05	1
R2.3	The color of the font will be Green.	Font Color Green	Medium	0.15	2

Table 29 Customer Requirements Variant 1 Version 1 (R1.1)

At the same time, the customer spins off a second variant of these requirements in which the font color is “Red” (see Table 30). Using the numbering scheme proposed in [LEHC99] and [HARN99c], this set of requirements is referred to as Variant 2, Version 2.

R Tag	Requirement text	Name	ReqPro Priority	AHP Priority	Risk
R1	When commanded, the software must display the text “Hello World” at the terminal.	Write Hello World	High	0.75	4
R2	The text will be pleasing to the user.	Pretty Font	Medium	0.0	2
R2.1	The font will be Arial.	Font	Low	0.05	1
R2.2	The font size will be 12.	Font Size	Low	0.05	1
R2.3	The color of the font will be Red.	Font Color Red	Medium	0.15	2

Table 30 Customer Requirements Variant 2 Version 2 (R2.2)

After reading through the requirements, the developer responds to the customer with a question related to potential implementation to which the customer responds (see Table 31).

Q Tag	Q&A	Name
Q1	Q: Is the use of a visual basic form good enough? A: yes.	Visual Basic Form

Table 31 Developer-Customer Question & Answer Version 1 & 2 (Q1.1, Q2.2)

The developer proceeds by developing a single SEATools model of the system (see Figure 74) that models both variants of the customer’s requirements.

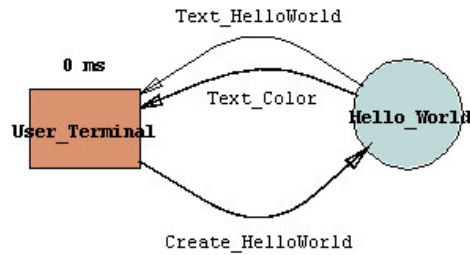


Figure 74 SEATools Hello World Prototype Variants 1 & 2, Versions 1 & 2

The developer uses the PSDL file from the SEATools model to make a list of the key specifications of the design based on the operators and data streams of the model. He also captures the “Required By” data for requirements traceability purposes. To each of these specifications, he assigns a level of difficulty (1= low difficulty, 9=high difficulty) based on his perception as the difficulty of fully implementing the particular specification into code. Table 32 lists these specifications.

S Tag	Specification type	Name	Difficulty	Required By
O1	Operator	Hello_World 16	0	R1
O1.1	Terminator	User_Terminal 19 18	2	R1, R2
O1.2	Operator	Hello_World 22 21	3	R1
E1	State Stream	Create_HelloWorld	1	R1
E2	Data Stream	Text_HelloWorld	1	R2, R2.1, R2.2
E3	State Stream	Text_Color	1	R2, R2.3

Table 32 Software Specifications Variants 1 & 2 Versions 1 & 2 (S1.1, S2.2)

The developer then assigns programmers to tackle particular portions of the application. The code hierarchy is similar for both variants and is shown in Table 33 and Table 34. Figure 75 provides screen shots of the actual implementation of both variants.

C Tag	Code Description	Name
C1	HCI	Form1
C1.1	Text settings (green)	Text1
C1.2	Command button	Command1

Table 33 Code Variant 1 Version 1 (C1.1)

C Tag	Code Description	Name
C1	HCI	Form1
C1.1	Text settings (red)	Text1
C1.2	Command button	Command1

Table 34 Code Variant 2 Version 2 (C2.2)



Figure 75 Hello World Implementation Variants 1 & 2, Versions 1 & 2

The developer provides the applications to a customer representative who performs validation testing of the variants against the originally stated customer requirements. The customer representative provides feedback as shown in Table 35.

F Tag	Feedback	Name
F1	The user should have the option to choose between Red or Green.	Color Choice
F2	The function of the command button is exactly what the user wants	Button works well
F3	The text size and font meet the customer's needs	Text Size & Font Good

Table 35 Customer Feedback Version 1 & 2 (F1.1, F2.2)

Based on feedback item “F1,” the customer restates his requirements and asks the developer to generate another version of the application (variant 2, version 3). The updated customer requirements (with new priority and risk values) are shown in Table 36.

R Tag	Requirement text	Name	ReqPro Priority	AHP Priority	Risk
R1	When commanded, the software must display the text "Hello World" at the terminal.	Write Hello World	High	0.527	4

R2	The text will be pleasing to the user.	Pretty Font	Low	0.0	2
R2.1	The font will be Arial.	Font	Low	0.094	1
R2.2	The font size will be 12.	Font Size	Low	0.046	1
R3	The user will have a choice of font color as either Green (default) or Red.	Font Color	Medium	0.333	4

Table 36 Customer Requirements Variant 2 Version 3 (R2.3)

The developer updates his Q&A list with “Q2” (see Table 37) and modifies his SEATools model to account for the new functionality (Figure 76).

Q Tag	Q&A	Name
Q1	Q: Is the use of a visual basic form, good enough? A: yes.	Visual Basic Form
Q2	Q: Is the use of radio buttons for choice, good enough? A: yes.	Radio buttons

Table 37 Developer-Customer Question & Answer Version 3 (Q2.3)

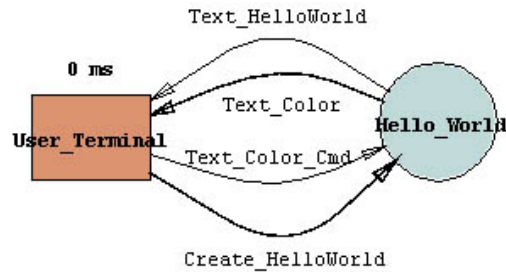


Figure 76 SEATools Hello World Prototype Variant 2, Version 3

The updated SEATools model leads to an updated list of software Specifications (see Table 38) with updated “difficulty” ratings.

S Tag	Specification type	Name	Difficulty	Required By
O1	Operator	Hello_World_16	0	R1
O1.1	Terminator	User_Terminal_19_18	2	R1, R2
O1.2	Operator	Hello_World_22_21	3	R1
E1	State Stream	Create_HelloWorld	1	R1
E2	Data Stream	Text_HelloWorld	1	R2, R2.1, R2.2
E3	State Stream	Text_Color	1	R3
E4	Data Stream	Text_Color_Cmd	3	R3

Table 38 Software Specifications Variant 2 Version 3 (S2.3)

The developer reassigns programmers to the coding effort. The new code hierarchy is shown in Table 39.

C Tag	Code Description	Name
C1	HCI	Form1
C1.1	Text settings	Text1
C1.2	Command button	Command1
C1.3	Radio button (default)	Option1_Green
C1.4	Radio button	Option2_Red

Table 39 Code Variant 2 Version 3 (C2.3)

Figure 77 illustrates the implemented variant/version. Note that in the center figure, no text color has been selected, yet “green” text color was displayed (i.e. a green default value was implemented).



Figure 77 Hello World Implementation Variant 2, Versions 3

The customer, satisfied with this final variant/version provides the feedback in Table 40.

F Tag	Feedback	Name
F1	Everything works will, ship the product	Meets all Rqts

Table 40 Customer Feedback Version 3 (F2.3)

The developer turns the application over to the customer. This ends the Hello World software development scenario.

2. “CARA Infusion Pump” Software Scenario

The second software development scenario conducted in the dissertation experiment comes from a real world software development problem associated with the Computer Assisted Resuscitation Algorithm (CARA) [WRAI01a]:

The Computer Assisted Resuscitation Algorithm (CARA) is a closed loop software system that drives a high output infusion pump (M100) used for fluid resuscitation of patients suffering from conditions that lead to hypotension. The system will use blood pressure as the control for a proportional closed-loop control algorithm. The CARA system will be ultimately fielded on up to 3 platforms: LSTAT (Life Support for Trauma and Transport), DataPak and WPSM, collectively known as HOSTS. The software will have to accommodate various blood pressure inputs (e.g. arterial line, noninvasive cuff, pulse wave, etc)... The CARA is intended to support primary intravenous fluid resuscitation therapy to rapidly restore intravascular volume and blood pressure in patients with clinical shock, hypotension, and hypoperfusion states as a result of hemorrhagic blood loss, occult hemorrhage, neurogenic shock and septic shock. Currently the uses for the CARA system will be for combat casualties.

In this software development scenario, five teams of developers were each given identical sets of CARA software requirements [WRAI01c] (also Appendix B) and a historical set of discussions between developers and customers (framed as 133 questions and answers) [WRAI01b]. These discussions often related directly to particular requirements in the requirement set and helped to clarify ambiguities in the requirements for the developers. A sample of these questions and answers is shown in Table 41.

Tag	Question	Title	Response	Reference Reqmt	Action
Q38	2/3/99 – Is the pause auto-control mode necessary?	AC Pause necessary	2/3/99 – No, it can be eliminated to simplify the system and reduce the operating hazards.	33, 35, 36, 37, 38, 40, 48.4.2	Remove 33, 35, 36, 37, 38, 40, 48.4.2
Q39	2/3/99 – Should 5 cuff measures be taken to calibrate the PW, or will fewer suffice?	Less than 5 readings to Calibrate Cuff BP	2/3/99 – The number of cuff measures required for calibration should be reduced to 3.	24.1	24.1
Q40	2/3/99 – Will PA or CVP be used for auto-control?	PA or CVP for AC	2/3/99 – No.		N/A
Q41	2/3/99 – When an action button is pressed does it remain available, or is it disabled?	Action Buttons	2/3/99 – The action button should be disabled (possibly removed) once the button has been pressed.		49

Table 41 Excerpt of CARA Questions and Answers (after [WRAI01b])

From these requirements and discussions, each of the five teams then independently constructed a SEATools model of the CARA software. The teams did not

see their particular development effort through to the completion of full functioning executable software systems, only to the specifications development stage (as embodied as a SEATools PSDL file). The software development process model for this scenario is illustrated in Figure 78.

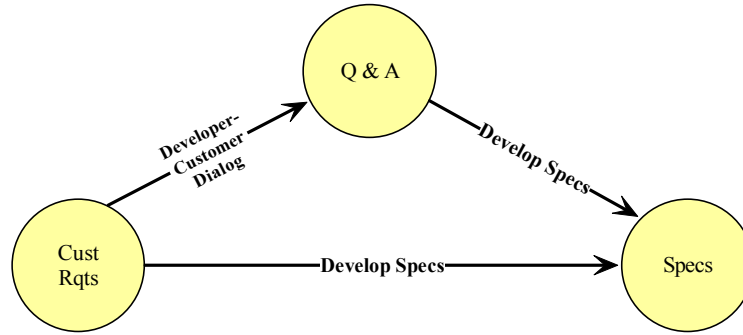


Figure 78 CARA Software Development Process

Note that this is an abbreviated software development process that concluded at the specifications development phase. The customer begins by stating his requirements in Requisite®Pro (see Appendix B). The customer identifies the priorities of his requirements using a user defined requirements attribute called “AHP Priority” and then using the Analytic Hierarchy Process (AHP) [SAAT80] to provide values for that priority. The customer also creates a second user defined requirements attribute called “Risk.” He assigns to risk, values associated with the clarity of requirements as measured by how frequently developers ask questions about that particular requirement. Requirements with low risk are those in which the developer asks no questions and those with high risk are those in which the developer asks multiple questions (1=low risk, 5=high risk). The third user-defined requirement’s attribute is “Safety” in which the user again uses a 1 to 5 scale to indicate the importance of that particular requirement to the safety criticality of the overall system design.

While all five SEATools models developed by the independent teams are described in detail and compared in [LUQI02] and [LUQI03], only one such model was used in the dissertation experiment and is explained in detail in Appendix C. Using the associated PSDL file of the graphic SEATools software model, a list of software development specifications was generated for the variant. This variant was numbered as 1.1 in accordance with the numbering scheme presented in [LEHC99] and [HARN99c].

An excerpt from the model 1 specifications is shown below in Table 42 (this portion of the specification relates to the IO Module described in Appendix C).

S Tag	Spec Type	Specification Name	Required by
O1.2.2	Operator	IO_Module	
O1.2.2.1	Operator	alarm_controller1	
O1.2.2.1.1	Operator	Alarm_Display_Generator	R6.1, R6.2, R7.1.1, R7.1.2, R8.1.1, R8.1.2, R15.1, R15.2
O1.2.2.2	Operator	alarm_controller2	
O1.2.2.3	Operator	button_monitor	
O1.2.2.4	Operator	display_driver	
O1.2.2.4.1	Operator	Pump_Plugged_in_Display_Driver	R2, R2.1, R2.2, R2.3, R2.4, R2.5, R12.1
O1.2.2.4.2	Operator	LSTAT_power_on_Display_driver	R1
O1.2.2.4.3	Operator	BP_Graph_and_Value_Driver	R14.1.3, R14.1.4
O1.2.2.4.4	Operator	CARA_Status_Display_Driver	R16, R16.1, R16.2
O1.2.2.4.5	Operator	Voume_Infused_and_Flow_Rate_Display_Driver	R10.2, R11.1, R11.1.1
O1.2.2.4.6	Operator	Immediate_Feeedback_Display_Driver	

Table 42 Excerpt from CARA Model 1, Specifications 1.1

While the specific step by step software development activities for each team varied somewhat by team, the general process followed by all teams consisted of analyzing the requirements, clarifying requirements by considering the developer-customer questions and answers and then in building a SEATools model that satisfied the set of requirements and was consistent with the questions and answers.

C. RESULTS AND CONFIRMING EVIDENCE OF THE HYPOTHESIS

Applying the HFSE to Rational's Requisite®Pro and SEATools, each of the software development scenarios was completed twice: once using tools that had not been integrated into the HFSE and once using tools that had been integrated with the HFSE.

1. "Hello World" Results

a. Questions Posed

While completing the Hello World scenarios, the following questions were asked at particular points in the development effort. This set of questions is not exhaustive and many similar questions could have been posed in order to highlight

additional benefits of the HFSE. The questions and the presence or absence of answers are illustrative of the improvements of interoperability (additional joint task execution) provided by the HFSE.

- (1) If a change is made to requirement 2.3 (variant 2 version 2), what other artifacts in the remainder of the development effort will likely change?
- (2) Which portion of the code poses the highest risk from a customer's standpoint?
- (3) Which requirements lead to the most difficult specification?
- (4) What are the most difficult portions of the code?
- (5) Which artifacts will need to change based on customer feedback F1(variant 2 version 2)?
- (6) Which pieces of code are the most important to the customer?
- (7) Are all requirements adequately covered by specifications?

b. Non-HFSE Results: Hello World

During completion of the non-HFSE scenario, none of the above questions could be answered quickly or reliably because there was no automated decision support information. Both the developer and customer could hazard guesses at each of these; unfortunately, there was no direct supporting analytical information for their answers.

c. HFSE Results: Hello World

During completion of the HFSE scenario, each of the above questions was answered reliably because there was automated decision support information available to the customer and developer. Using CASESv2 as the HFSE support tool, a project schema (Figure 79 below) was established that mirrored the software development process of development effort (recall Figure 73).

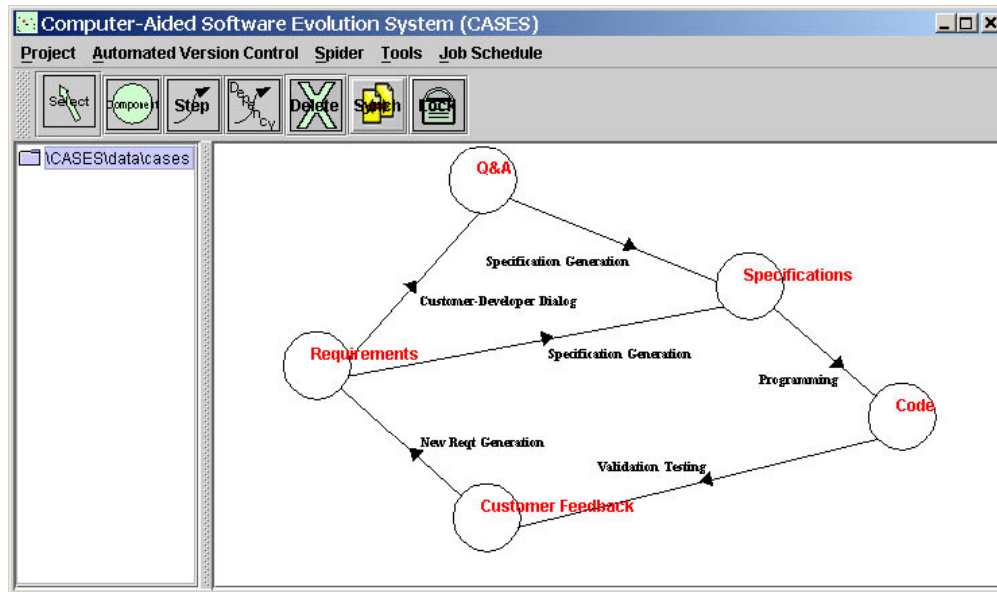


Figure 79 Hello World Project Schema

Table 29 through Table 40 were used as CSV input to the project schema. Note that the “PSDL Model” and the actual application “Implementation” are not included as separate artifacts in the project schema since their information is actually embodied in the “Specifications” and “Code” artifacts, respectively. Dependency relationships were established for “ReqPro Priority,” “AHP Priority,” “Risk,” and “Difficulty.” Throughout the development effort, QFD correlations were established as appropriate between all components. As QFD correlations were completed, dependency calculations were synchronized, thus “deploying” the dependencies throughout the development effort. This deployment provided ready decision support information for answering the questions. Discussion of this automated decision support information follows.

(1) Question 1 was, “If a change is made to requirement 2.3 (variant 2 version 2), what other artifacts in the remainder of the development effort will likely change?” The answer was obtained by performing a “Component Trace” with threshold “1” on Requirement “2.3, variant 2, version 2” (see Figure 80).

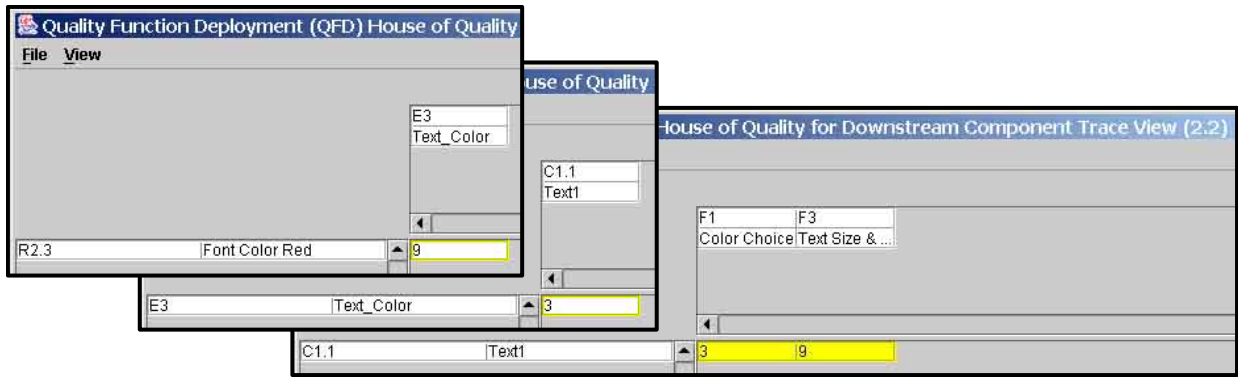


Figure 80 Component Trace: R2.2-2.3, $t = 1$

The answer included the following components: S2.2-E3, C2.2-1.1, F2.2-1, and F2.2-3. The implication to the developer was that given this information, he could make a more accurate assessment of costs associated with making the single change to that requirement. While this specific decision support information only provided the number and type of components that would likely change, the developer could view particular QFD matrices with the “Difficulty” dependency of these components to obtain accurate information about the relative difficulty each of changing each of these components.

(2) Question 2 was, “Which portion of the code poses the highest risk from a customer’s standpoint?” The answers to this question were obtained in less than 3 seconds for each variant by simply viewing the QFD Matrix “Risk Dependency” between Specifications and Code (see Figure 81).

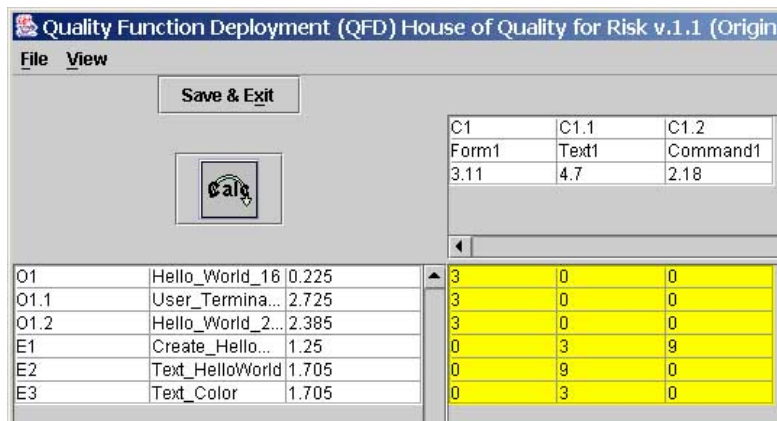


Figure 81 QFD Matrix: S1.1 x C1.1, $d = \text{Risk}$

For Variant 1 and 2, version 1 and 2, the answer was the C1.1 “Text1” implementation piece of code with a risk value of 4.7. For Variant 2 version 3, the answer was the C1.3

and C1.4 pieces of code (implementation of the two radio buttons) with a risk value of 3.65 (this make sense give that this was the new piece of functionality that was being added to the existing design). The implication to the developer was he should probably assign one of his better programmers to these pieces of code since they represented the highest level of risk to the customer. Incidentally, the same pieces of code (for variant 2 version 3) had the highest level of priority from the user and had the highest level of difficulty from the developer. Together, these pieces of information allowed the developer to make an informed choice as to what resources to apply to this particular section of the code.

(3) Question 3 was, “Which requirements lead to the most difficult specification?” The answer was obtained by opening the Requirement x Specification QFD Matrix for the Difficulty Dependency, then performing a “Component Trace” with threshold “1” on specification “O1.2” (which has the highest difficulty value of “3”) (see Figure 82).

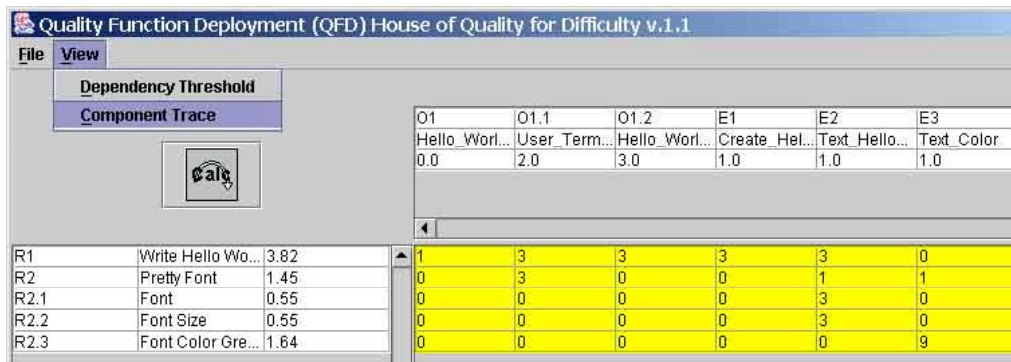


Figure 82 Component Trace: S1.1-O1.2, $t = 1$

The answer included the following component R1.1-1 (for variant 1 and 2, version 1 and 2). In the case of variant 2 version 3, there where two specifications each with a level of difficulty of 3, therefore two separate traces were required which led to two requirements: R2.3-1 and R2.3-3. The implication to the developer was that should he run into trouble in implementing this relatively more difficult specification, he would know exactly which requirements would be affected. Note that this particular question could have simply been answered by considering the O1.2 column of the correlation matrix and looking across to the left

(4) Question 4 was, “What are the most difficult portions of the code?” The answers to this question was obtained by simply opening the QFD Matrix “Difficulty Dependency” between Specifications and Code and obtaining a “Dependency Threshold” view with the threshold equal to the mean (see Figure 83).

Specification	Code	Difficulty	Threshold
O1.1	User_Termina...	2.0	3
O1.2	Hello_World_2...	3.0	0

Figure 83 Dependency Threshold: S1.1 x C1.1, d = Difficulty, $t = \mu$

For Variant 1 and 2, version 1 and 2, the answer were the C1 “Form1” and C1.1 “Text1” pieces of code with a difficulty value of 3.08 each. For Variant 2 version 3, the answer was the C1.3 and C1.4 pieces of code (the color radio buttons) with a difficulty value of 3.24. The implication to the developer was he should probably assign one of his better programmers to these pieces of code and perhaps should subject these pieces of code to more rigorous testing than other portions of the code. Once again, these QFD dependency information has allowed the developer to make an informed choice as to what resources to apply to these particular sections of the code.

(5) Question 5 was, “Which artifacts will need to change based on customer feedback F1 (variant 2 version 2)?” The answer was obtained by performing a “Component Trace” with threshold “1” on Feedback “F1, variant 2, version 2”. The resulting answer presented the developer with a set of components that included almost the entire underlying hypergraph meaning that almost every component might require change. The developer then modified his query and changed the threshold to 3 (see Figure 84).

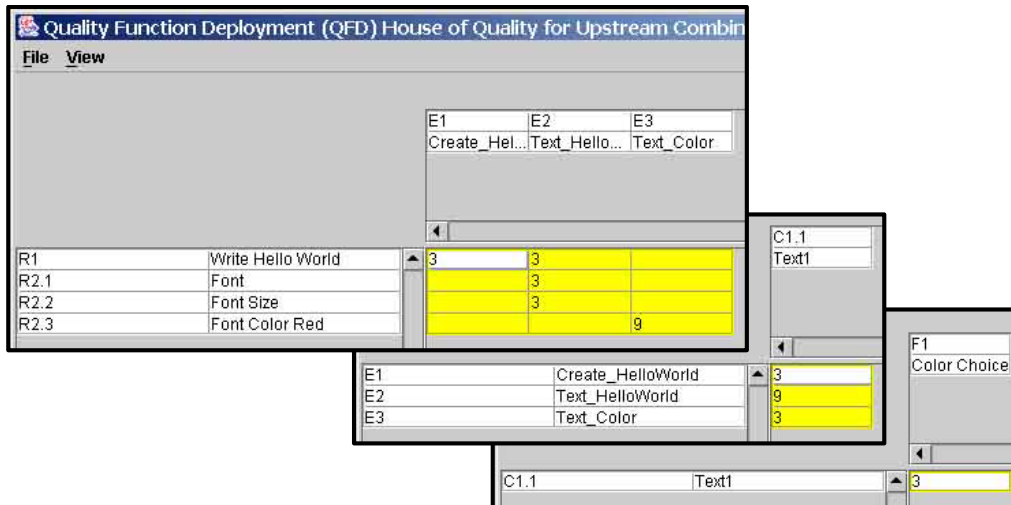


Figure 84 Component Trace: F2.2-1, $t = 3$

The answer included the following components C2.2-1.1; S2.2-E1, E2, E3; R2.2-1, 2.1, 2.2, and 2.3. The implication to the developer was that given this information, he could make a more accurate assessment of costs associated with modifying the development artifacts in addressing that particular piece of feedback. While this specific decision support information only provided the number and type of components that would likely change, the developer could view particular QFD matrices with the “Difficulty” dependency of these components to obtain accurate information about the relative difficulty each of changing each of these components.

(6) Question 6 was, “Which pieces of code are the most important to the customer?” The answers to this question was obtained by opening the QFD Matrix “AHP Priority Dependency” between Specifications and Code and obtaining a “Dependency Threshold” view with the threshold equal to the mean plus 0.5 standard deviations (see Figure 85).

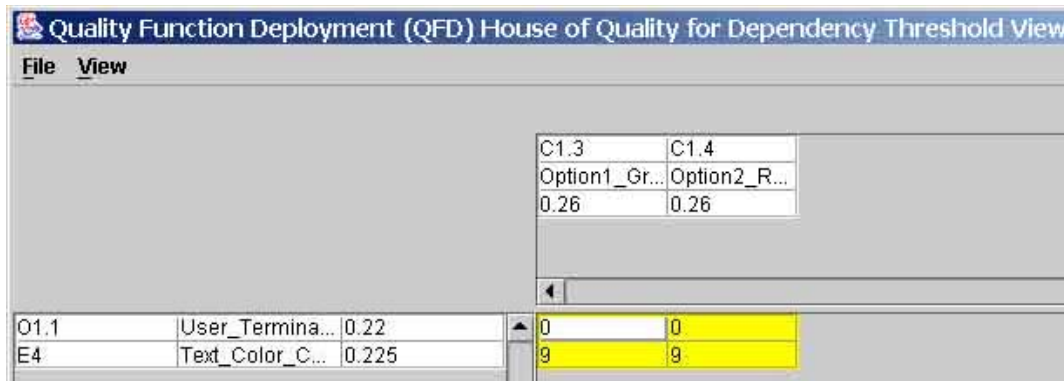


Figure 85 Dependency Threshold: S2.3 x C2.3, $d = \text{AHP Priority}$, $t = \mu + 0.5\sigma$

For Variant 1 and 2, version 1 and 2, the answer was the C1.1 “Text1” piece of code with an AHP Priority value of 0.44. For Variant 2 version 3, the answer was the C1.3 and C1.4 pieces of code (the radio button features) with an AHP Priority value of 0.26. The implication to the developer was he should probably assign one of his better programmers to these pieces of code and perhaps should subject these pieces of code to more rigorous testing than other portions of the code. Once again, this QFD dependency information has allowed the developer to make an informed choice as to what resources to apply to these particular sections of the code.

(7) Question 7 was, “Are all requirements adequately covered by specifications?” The answers to this question was obtained by opening the QFD Matrix “Priority Dependency” between Requirements and Specifications and looking at the QFD correlation matrix (see Figure 86) to see if there were sufficient correlations for each requirement. While CASESv2.0 does not directly support the coverage calculation (delineated in Chapter IV), the developer could make a quick estimate by reading across the rows of the QFD matrix and ensuring that each high Priority valued requirement had relatively more total correlation than low Priority valued requirements.

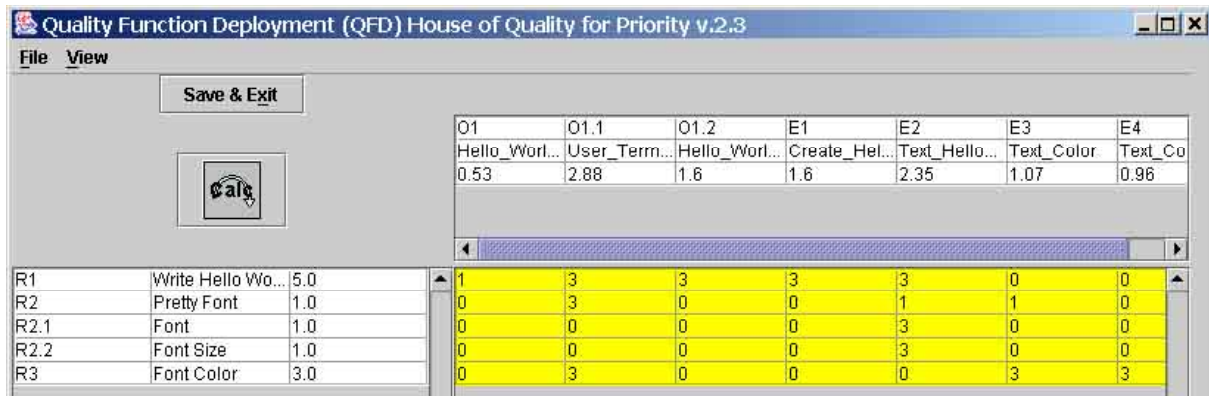


Figure 86 QFD Matrix: R2.3 x S2.3, d = Priority

For Variant 1 and 2, version 1 and 2, the answer was yes, the requirements were adequately covered by specifications. For Variant 2 version 3, the answer was also yes, the requirements were adequately covered by specifications. The implication to the developer was he did not need to develop any additional specifications to ensure that the customer's requirements were adequately addressed.

2. "CARA Infusion Pump" Software Scenario

a. Questions Posed

After completing the CARA development scenario, the following questions were asked at the end of the development effort. This set of questions is not exhaustive and many similar questions could have been posed in order to highlight the benefits of the HFSE. The questions and the presence or absence of answers are illustrative of the improvements of interoperability (additional joint task execution) provided by the HFSE.

- (1) Which elements of the SEATools models are the most safety critical?
- (2) Which elements of the SEATools models are the most important to the customer?
- (3) Which elements of the SEATools models are the most risky (based on Requirements Clarity) from a customer perspective?
- (4) If requirement R34 were modified, which portions of the SEATools model would have to be modified?

(5) In Model 1, which questions/answers and requirements are related to the “Resuscitation Log”?

(6) In Model 1, which questions/answers and requirements are related to the “Triple Modular Redundancy” feature of the design?

(7) In Model 1, which questions/answers and requirements are related to the “Processor Watchdog” feature of the design?

b. Non-HFSE Results: CARA

After completion of the non-HFSE scenario, none of the above questions could be answered quickly or reliably because there was no automated decision support information. Both the developer and customer could hazard guesses at each of these; unfortunately, there was no direct supporting analytical information for their answers.

c. HFSE Results: CARA

After completion of the non-HFSE scenario, the Requirements, Questions/Answers, and SEATools models were integrated via the HFSE and CASESv2.0. Each of the above questions was then answered quickly and reliably because there was automated decision support information available to the customer and developer. Using CASESv2 as the HFSE support tool, a project schema similar to that of Figure 78 was established. Only model 1 (variant 1.1) of the five development variants was defined within CASES. The Requirements CSV file (similar to appendix B), the Q&A CSV file (similar to Table 41), and a SEATools Specifications file for the variant (similar to Table 42) were imported into the project schema. Dependency relationships was established for “AHP Priority,” “Safety” and “Risk.” QFD correlations were established as appropriate between all components. As QFD correlations were completed, dependency calculations were synchronized, thus “deploying” the dependencies throughout the development effort. This deployment provided ready decision support information for answering the questions. Discussion of this automated decision support information follows.

(1) Question 1 was, “Which elements of the SEATools models are the most safety critical?” The answer was obtained by opening the QFD Matrix “Safety Dependency” between Requirements and Specifications and generating a

“Dependency Threshold” view with the threshold equal to the mean plus 1.5 standard deviations (see Figure 87).

Component	01.2.1.1.1.3 BP Priority...	01.2.2.1.1 Alarm Dis...	01.2.2.3 button_mo...	01.2.2.5.1 Display Al...	E15 lost_bp so...	E28 display ala...
01.2.1.1.1.3 BP Priority...	36.85					
01.2.2.1.1 Alarm Dis...		35.48				
01.2.2.3 button_mo...			24.49			
01.2.2.5.1 Display Al...				31.9		
E15 lost_bp so...					20.24	
E28 display ala...						26.17

Feature	01.2.1.1.1.3 BP Priority...	01.2.2.1.1 Alarm Dis...	01.2.2.3 button_mo...	01.2.2.5.1 Display Al...	E15 lost_bp so...	E28 display ala...
FEAT6.3 Discontinuity T...	0.0	0.0	0.0	0.0	0.0	0.0
FEAT7.1.3 Occlusion Ter...	0.0	0.0	0.0	0.0	0.0	0.0
FEAT8.1.3 AirOK Termina...	0.0	0.0	0.0	0.0	0.0	0.0
FEAT11.2.2 EMF Unobtain...	0.0	0.0	0.0	0.0	0.0	0.0
FEAT17.6 No Available B...	1.0	0.0	0.0	3.0	0.0	0.0
FEAT29 Calculate Pum...	3.0	0.0	0.0	0.0	0.0	0.0
FEAT31 Minimum Flow...	9.0	0.0	0.0	0.0	0.0	0.0
FEAT34 Terminate AC ...	1.0	0.0	9.0	0.0	0.0	0.0
FEAT43.3 Beat-to-Beat Si...	9.0	0.0	0.0	0.0	0.0	0.0
FEAT44.3.1.3 If New Cuff BP ...	9.0	0.0	0.0	0.0	0.0	0.0
FEAT47.3.3 Pump Unplug...	0.0	0.0	0.0	0.0	0.0	0.0
FEAT48 Terminate AC ...	0.0	0.0	9.0	0.0	0.0	0.0
FEAT48.3.1 Terminate AC ...	0.0	0.0	9.0	0.0	0.0	0.0
FEAT54.1.1 No Impedance...	0.0	0.0	0.0	0.0	0.0	0.0

Figure 87 Dependency Threshold: R1.1 x S1.1, $d = \text{Safety}$, $t = \mu + 1.5\sigma$

The answer included the following SEATools specifications shown in Table 43.

Variant/ Version	Components
1.1	01.2.1.1.1.3 BP_Priority_Calculator, 01.2.2.1.1 Alarm_Display_Generator, 01.2.2.3 button_monitor, 01.2.2.5.1 Display_Alarm, E15 lost_bp_source, E28 display_alarm_data

Table 43 Most Safety Critical Components: R x S, $d = \text{Safety}$, $t = \mu + 1.5\sigma$

The developer could have increased or decreased the number of components in the answer by modifying the threshold. The implication of this information to the developer was that he could latter design more exhaustive testing of these particular components or perhaps he might choose to group the implementation of these specifications into distinct modules to which he would apply formal methods. This information allows the developer to make informed choices about how and where to apply limited development resources.

(2) Question 2 was, “Which elements of the SEATools models are the most important to the customer?” The answer was obtained by opening the QFD Matrix “AHP Priority Dependency” between Requirements and Specifications and

generating a “Dependency Threshold” view with the threshold equal to the mean plus 2.5 standard deviations (see Figure 88)

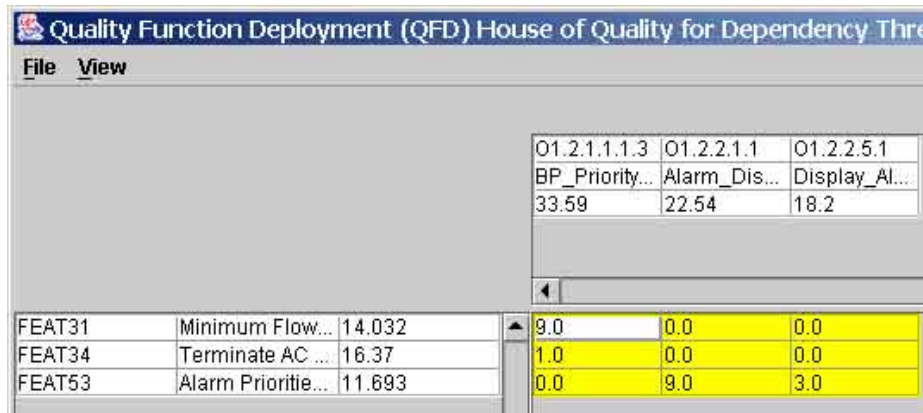


Figure 88 Dependency Threshold: R1.1 x S1.1, $d = \text{AHP Priority}$, $t = \mu + 2.5\sigma$

The answer included the following components shown in Table 44.

Variant/ Version	Components
1.1	O1.2.1.1.1.3 BP_Priority_Calculator, O1.2.2.1.1 Alarm_Display_Generator, O1.2.2.5.1 Display_Alarm

Table 44 Most Important Components: R x S, $d = \text{AHP Priority}$, $t = \mu + 2.0\sigma$

Once again, the developer could have trimmed or expanded the number of components found by modifying the threshold of the search. The implication to the developer was he might want to assign one of his better programmers to these specifications and perhaps should subject the pieces of code generated from these specifications to additional validation testing to ensure that they meet or exceed customer’s expectations. Once again, this QFD dependency information has allowed the developer to make an informed choice as to what resources to apply to the development effort.

(3) Question 3 was, “Which elements of the SEATools models are the most risky (based on Requirements Clarity) from a customer perspective?” The answer was obtained by opening the QFD Matrix “Risk Dependency” between Requirements and Specifications and generating a “Dependency Threshold” view with the threshold equal to the mean plus 1.0 standard deviations (see Figure 88)

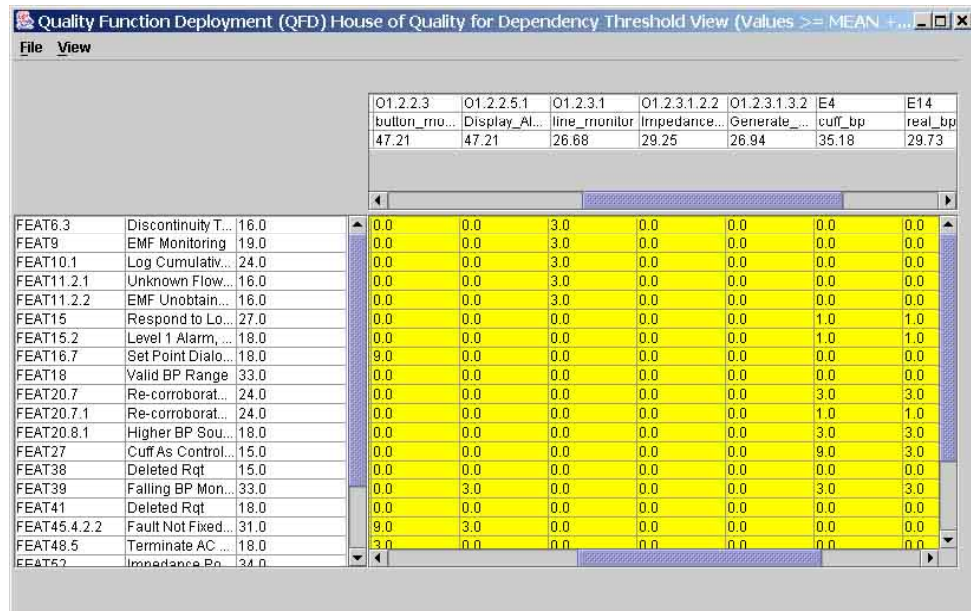


Figure 89 Dependency Threshold: R1.1 x S1.1, $d = \text{Risk}$, $t = \mu + 1.0\sigma$

The answer included the components shown in Table 45.

Variant/ Version	Components
1.1	O1.2.1.1.1 BP_calculator, O1.2.1.1.1.1 Aline_Corroborator, O1.2.1.1.1.3 BP_Priority_Calculator, O1.2.1.4.2 Terminate_autocontrol, O1.2.2.1.1 Alarm_Display_Generator, O1.2.2.3 button_monitor, O1.2.2.5.1 Display_Alarm, O1.2.3.1 line_monitor, O1.2.3.1.2.2 Impedance_Calculator, O1.2.3.1.3.2 Generate_Air_fault, E4 cuff_bp, E14 real_bp, E15 lost_bp_source, E28 display_alarm_data

Table 45 Most Risky Components: R x S, $d = \text{Risk}$, $t = \mu + 1.0\sigma$

Once again, the developer could have trimmed or expanded the number of components found by modifying the threshold of the search. The implication to the developer was that these specifications were directly traceable to requirements for which many questions were asked, an indication that the requirements were not as well understood or clear as other requirements. Again, the developer might want to undertake some form of mitigation (such as additional testing, or use his better programmers) in support of these particular specifications.

(4) Question 4 was, “If requirement R34 were modified, which portions of the SEATools model would have to be modified?” The answer was obtained

by performing a “Component Trace” with threshold “3” on Requirement “34” (see Figure 90).

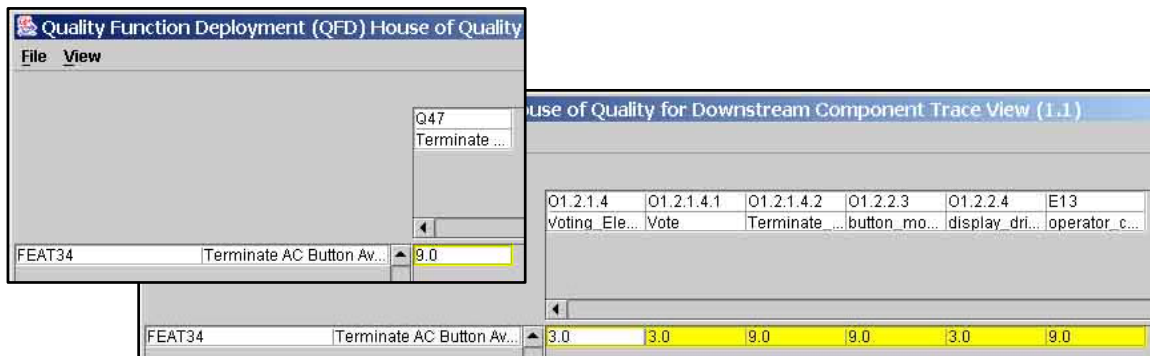


Figure 90 Component Trace: R1.1-34, $t = 3$

The answer included the following components shown in Table 46.

Variant/ Version	Components
1.1	Q47 Terminate AC button, O1.2.1.4 Voting_Element, O1.2.1.4.1 Vote, O1.2.1.4.2 Terminate_autocontrol, O1.2.2.3 button_monitor, O1.2.2.4 display_driver, E13 operator_commands

Table 46 Component Trace: R1.1-34, $t = 1$

The developer could have gained visibility over more or fewer components by varying the degree of the threshold of correlation (“3” was used here). The implication to the developer was that given this information, he could make a more accurate assessment of costs associated with making the single change to that requirement.

(5) Question 5 was, “In Model 1, which questions/answers and requirements are related to the ‘Resuscitation Log?’” The answer was obtained by performing a “Component Trace” with threshold “1” on Specifications that were related to the resuscitation log. In the case of model 1, this included three specifications: “O1.2.3.5 resuscitation_file operator”, “O1.2.3.5.9 Resuscitation_file_Generator operator,” and “E38 resuscitation_file data stream.” The Trace of the first specification is shown in Figure 91.

Feature	Value
FEAT5	3.0
FEAT10.1	3.0
FEAT11.2	3.0
FEAT11.2.1	3.0
FEAT12	3.0
FEAT14	1.0
FEAT14.1	1.0
FEAT14.1.1	1.0
FEAT14.1.2	1.0
FEAT15.3	1.0
FEAT16.5.4	1.0
FEAT17.1	1.0
FEAT17.3.3.2	1.0
FEAT17.4	1.0
FEAT17.5	1.0
FEAT17.6.2	1.0
FEAT20.3.2	1.0
FEAT20.6	1.0
FEAT29.7	1.0

Figure 91 Component Trace: S1.1-O1.2.3.5, $t = 1$

The answer included the following components shown in Table 47.

Trace	Components
O1.2.3.5	Features (FEAT) 5, 10.1, 11.2, 11.2.1, 12, 14, 14.1, 14.1.1, 14.1.2, 15.3, 16.5.4, 17.1, 17.3.3.2, 17.4, 17.5, 17.6.2, 20.3.2, 20.6, 29.2, 44.4, 46, 46.2, 47, 48.3.1.2, 52.1, 55, 56, 62, 66, 69
O1.2.3.5.9	Features (FEAT) 5, 10.1, 11.2, 11.2.1, 12, 15.3, 16.5.4, 17.3.3.2, 17.4, 17.5, 17.6.2, 20.3.2, 20.6, 29.2, 44.4, 46, 46.1, 46.2, 47, 48.3.1.2, 55, 56, 62, 66, 69
E38	Features (FEAT) 5, 10.2, 11.2, 11.2.1, 12, 14, 14.1, 14.1.1, 14.1.2, 15.3, 16.5.4, 17.3.3.2, 17.4, 17.5, 17.6.2, 20.3.2, 20.6, 29.2, 44.4, 46, 46.1, 46.2, 47, 48.3.1.2, 52.1, 55, 56, 62, 66, 69

Table 47 Component Trace: S1.1-O1.2.3.5, O1.2.3.5.9, E38, $t = 1$

The implication to the developer was that he could use the information to ensure his specification met all requirements associated with the resuscitation log. Alternatively, if changes were needed for the resuscitation log specification, he would know which requirements might need to be reexamined to ensure that the modified resuscitation log specification remained consistent with already specified requirements.

(6) Question 6 was, “In Model 1, which questions/answers and requirements are related to the ‘Triple Modular Redundancy’ feature of the design?” The answer was obtained by performing a “Component Trace” with threshold “1” on the specifications related to Triple Modular Redundancy. In the case of model 1, two components implement this feature: “O1.2.1.2 Module2 operator” and “O1.2.1.3 Module3 operator.” The answer was that no requirements or Q&As were related to this specification. The implication to the developer was that he had generated specifications for which there were no stated requirements. In this particular case, the developer could query the customer to determine if such functionality was needed or desired, or he could delete these specifications from the design, or he could choose to implement the specifications despite not having a requirements induced rationale for the specifications. In any case, the developer now has visibility over important design (and safety) information that was not available to him during the non-HFSE scenario.

(7) Question 7 was, “In Model 1, which questions/answers and requirements are related to the ‘Processor Watchdog’ feature of the design?” The answer was obtained by performing a “Component Trace” with threshold “1” on the specifications related to the processor watchdog. This included three specifications in Model 1: “O1.2.3.3 processor_watchdog operator,” “E20 ping data stream,” and “E21 acknowledgement data stream.” Much like question (6), the answer was that no requirements or Q&As were related to this specification. Once again, the implication to the developer was that he had generated specifications for which there were no stated requirements and as in the case of questions (6) the developer could query the customer to determine if such functionality was needed or desired, or he could delete these specifications from the design, or he could choose to implement the specifications despite not having a requirements induced rationale for the specifications.

3. Evidence Confirming the Dissertation Hypothesis

As illustrated above, in both development scenarios confirming evidence of the dissertation hypothesis was established. The HFSE provided improvements in joint task execution in both scenarios. These improvements are illustrated by the information jointly generated by the subordinate development tools in response to specific questions about the development effort. While only seven questions were asked in each

development effort, many more such questions could have been asked in order to provide quantitatively more evidence; however, qualitatively, this evidence would have been similar to that generated by the above set of questions.

D. INTERNAL AND EXTERNAL VALIDITY AND EXPERIMENTAL IMPLICATIONS

1. Internal and External Validity

Campbell and Stanley [CAMP63] lay out the conditions for which scientifically sound experimentation should occur. In order for an experiment to be scientifically sound, the experiment must bound sources of internal and external invalidity. Internal validity deals with the question of whether or not the application of the process (X) was, in fact, the sole direct contributing cause of the measured result. External validity deals with the question of whether the result can be generalized to external populations, sets, etc. outside the experiment. The static group comparison that gathered confirming evidence in this dissertation controls some (but not all) of these sources of invalidity.

2. Sources of Internal Invalidity

a. History

This source of internal invalidity arises because of specific events occurring between measurements of the outcome that are in addition to the experimental variable. This source was not completely controlled during the experiment because once the tool set (O) was integrated by the HFSE, its state did change during the time period of the search for evidence of improvements in interoperability of (O). The state changed because the experiment required seeking interoperability improvements during an active evolutionary software development effort (albeit a very limited development effort). While unlikely to be the cause, these state changes cannot be ruled out as the effect of additional improvements in interoperability. The only way to have controlled this source of invalidity would have been to repeatedly apply the HFSE to (O) after documenting each improvement in interoperability (i.e. start from scratch each time). Given that the main interest was in establishing evidence in a complex evolutionary system, such an approach was impractical. Instead, the approach adopted to mitigate this source of

invalidity was to determine and document the direct cause of each improvement to interoperability.

The implications of “History” as a source of invalidity on the experiment appear to be negligible. At no time were any of the improvements to interoperability traceable to state changes within the integrated toolset. However, as stated above, this source of invalidity, while mitigated, could not be entirely ruled out.

b. Maturation

This source of internal invalidity arises because of processes within (O) that may change as a function of time, independent of any application of the HFSE (X). This source of invalidity was controlled in the conduct of this experiment but cannot be ruled out if others attempt to repeat the experiment using a dynamic middleware HFSE communications mechanism rather than the static middleware mechanism used in this dissertation (i.e. importing static CSV files). Because software development process tools may have internal processes that are activated solely by time (e.g. automatic updating/rectifying of databases) a dynamic middleware solution might create changes within CASES that could possibly provide interoperability improvements not related to the application of the HFSE. Such processes would change the state of (O), meaning that it would not be possible to establish that the direct cause of differences in (O) were a result of the HFSE (X). Fortunately, this was not the case in this experiment.

Because of the use of a static middleware mechanism between active software development tools and CASES, it was not possible for “Maturation” to enter the experiment as a potential source of invalidity. Therefore, there is high confidence that the improvements in interoperability uncovered during the software development scenarios were not caused by maturation.

c. Testing

This source of internal invalidity arises when the act of taking an observation changes the state of the observed item and thus influences future observations. This source of invalidity was adequately controlled since observing the evidence of improved interoperability within CASES did not generate any changes to the state of the relationships stored within CASES. In all each of the scenarios “views” of

existing data were obtained to provide confirming evidence. The underlying data remained unchanged.

The implication is that there is high confidence that the improvements in interoperability uncovered during the software development scenarios were not caused by a “Testing” source of invalidity.

d. Instrumentation

This source of internal invalidity arises because of changes in the observing instrument or changes in the observers create a bias between measurements. This source of invalidity could have arisen in this experiment since the experiment was seeking "improvements in interoperability" -- perhaps influenced by subjective opinion. The key for controlling this source of invalidity was to carefully define what an "improvement" means, to define what "interoperability" means, and to uniformly apply these definitions to the comparison set. As explained at the beginning of this chapter, the definitions used were as follows:

- Interoperability: data exchange and/or joint task execution between two or more separate tools/models.
- Improvement: evidence of the existence of interoperability found in the integrated tool/model set (type, quantity, speed), not found in the disjoint tool/model set.

These definitions proved sufficient to explain or account for phenomena witnessed during the experiment; however, because of the numerous variables involved, it is not guaranteed that these definitions will prove adequate should others attempt to repeat the results.

The implication is that “Instrumentation” was appropriately mitigated during the experiment, but cannot be ruled out in future experiments as others attempt to repeat these results.

e. Statistical Regression

This source of internal invalidity arises when the observation sample group has been selected from the extremes of the potential observation population. Since the tools/models selected for integration are more appropriately termed a "convenience" sample, this source of invalidity was not applicable to this experiment and therefore was controlled.

The implication is that there is high confidence that the improvements in interoperability uncovered during the software development scenarios were not caused by a “Statistical Regression” source of invalidity.

f. Selection Biases

This source of internal invalidity arises because of biases in the selection of the observation group (O). As mentioned in above, the observation group of tools integrated by the HFSE is best termed a convenience sample, chosen primarily on the basis of the availability of the tool/model. Two of the tools/models (SEATools and CASES) were specifically developed here at the Naval Postgraduate School with a view that they could be eventually integrated. Because these tools/models were hand-picked (SEATools as a participant in the observation group and CASES as the support tool of the HFSE), this selection forms a bias and this source of invalidity was definitely present in the experiment and therefore was not controlled. To mitigate this somewhat, an outside, mainstream, commercial software development product (Rationale's Requisite Pro) was chosen as the requirements engineering tool. Also as a mitigation measure, each of the examples of joint task execution was considered in light of this particular source of invalidity. No evidence was found (but that does not mean it does not exist) that would indicate that this source was in fact the generative cause of the improvement in interoperability.

The implication is that the improvements in interoperability uncovered during the software development scenarios could have been caused by a “Selection Bias” source of invalidity and not by application of the HFSE to the observation group. The best way to overcome this limitation would be to undertake as future research, additional experiments in which other tools (non-NPS developed research software development tools) are chosen for integration in the HFSE. Just such an experiment is proposed in the “Future Research” section of Chapter IX.

g. Experimental Mortality

This source of internal invalidity arises when there is a loss of part of the observation group during the experiment. Application of the HFSE did not result in the loss of any portion of the tools in the observation group; therefore, this source of invalidity did not occur and therefore was controlled.

The implication is that there is high confidence that the improvements in interoperability uncovered during the software development scenarios were not caused by an “Experimental Mortality” source of invalidity.

h. Selection-Maturation Interaction.

This source of internal invalidity arises in multi-observation group experiments when interaction between the observation groups is mistaken for the effect of the experimental variable. Since this experiment did not involve the interaction of the two observation groups, it cannot occur and therefore was controlled.

The implication is that there is high confidence that the improvements in interoperability uncovered during the software development scenarios were not caused by a “Selection-Maturation Interaction” source of invalidity.

3. Sources of External Invalidity

a. Interaction of Testing and X

This source of external invalidity arises when a pretest might change the observation groups' responsiveness to the experimental variable. In this case, no pretest was applied to the observation group, therefore there is no opportunity for bias. Thus, this source of invalidity was controlled.

The implication is that there is high confidence that a “Selection-Maturation Interaction” source of invalidity was not present in the experiment. Thus, from this particular perspective (Selection-Maturation Interaction), the improvements in interoperability uncovered during the software development scenarios could be repeated in experiments in which other randomly chosen software development tools are integrated via the HFSE.

b. Interaction of Selection and X

This source of external invalidity arises because of interaction effects between the selected observation group and the experimental variable. In this case, the experimental variable (X) is the application of the HFSE, which has as its core the Hypergraph Evolution Model integrated in the CASES support tool. This model and SEATools were originally designed to work together. Therefore, their interaction may have unfairly biased the generality of the result. Thus, this source of external validity

was not controlled and brings into question the practical application of the HFSE on other, randomly selected tools/models.

The implication is that there is low confidence that there was an absence of a “Selection-Maturation Interaction” source of invalidity in the experiment. Thus, from this particular perspective (Interaction of Selection and X), the improvements in interoperability uncovered during the software development scenarios might not be able to be repeated in experiments in which other randomly chosen software development tools are integrated via the HFSE. The best way to determine this is to undertake such an experiment (see Chapter IX, Future Research).

c. Reactive Arrangements

This source of external invalidity arises because of the reactive results of experimental arrangements. For instance, if all the tools chosen for the experiment all had a particular type of API it is not valid to conclude that the result of the experiment is generally applicable to all tools (for instance, those without APIs). Such a situation did exist in this experiment because there are many types of APIs (an essential aspect of how the individual tool ontologies were integrated) but only two such interfaces were used (the COM API of Requisite®Pro and the static PSDL output file of SEATools). Thus, this source of invalidity was not controlled. It is worth noting, though, that during conduct of the experiment there did not seem to be any extraordinary measures needed in order to integrate the two available APIs.

Therefore, there is low confidence that there was an absence of a “Reactive Arrangement” source of invalidity in the experiment. Thus, from this particular perspective (Reactive Arrangements), the improvements in interoperability uncovered during the software development scenarios might not be able to be repeated in experiments in which other randomly chosen software development tools are integrated via the HFSE. The best way to determine this is to undertake many such experiments testing as many different types of APIs as possible (see Chapter IX, Future Research).

d. Multiple-X Interference

This source of external invalidity arises when there are multiple treatments of the experimental variable on the same observation group. Since the HFSE was only

applied once to integrate Requisite®Pro and SEATools, there was no opportunity that this source of invalidity occurred. Thus, it was controlled.

The implication is that there is high confidence that a “Multiple-X Interference” source of invalidity was not present in the experiment. Thus, from this particular perspective (Multiple-X Interference), the improvements in interoperability uncovered during the software development scenarios could be repeated in experiments in which other randomly chosen software development tools are integrated via the HFSE.

4. Summary of Experimental Validity

Table 48 (below) summarizes the sources of invalidity associated with the proposed experiment. It is evident, that even with mitigation measures that there are a number of sources of invalidity. Because of the scope of this research, these sources of invalidity were not formally mitigated (only discussed). However, in Chapter IX Future Research, additional experiments that would directly address and account for these shortcomings are presented.

Sources of Internal Invalidity	History	~
	Maturation	Y
	Testing	Y
	Instrumentation	~
	Regression	Y
	Selection	X
	Mortality	Y
	Interaction of Selection & Maturation	Y
Sources of External Invalidity	Interaction of Testing and X	Y
	Interaction of Selection and X	X
	Reactive Arrangements	X
	Multiple-X interference	Y

Legend:

Y the source was controlled

X the source was not controlled

~ the source was mitigated

Table 48 Summary of Sources of Invalidity

The summarized results of Table 48 succinctly illustrate the strengths and weaknesses of the experiment. In terms of internal validity, each source was either totally controlled or adequately mitigated with the exception of “Selection.” Thus, confidence is high (but not necessarily conclusive) that application of HFSE and not some other internal source was in fact the cause of the improvements in interoperability observed in the experiment. To conclusively demonstrate that the HFSE is the sole cause of the improvement to interoperability, additional experiments in which randomly chosen tools are integrated into the HFSE should be undertaken. Such an experiment is proposed as future research.

The summarized results for external validity are not as strong as those for internal validity. The potential presence of “Interaction of Selection and X” and “Reactive Arrangements” sources of external invalidity cast doubt on the generality of the result.

The best way to prove this generality would be to continue to perform multiple similar experiments using randomly chosen sets of tools. Such experiments are proposed as “Future Research”.

E. CHAPTER SUMMARY

This chapter presented the validation approach used in the dissertation to obtain evidence confirming the dissertation hypothesis. The experiments used to obtain the confirming evidence were presented and explained. The confirming evidence of improved software development tool interoperability was identified. The sources of invalidity of the experiment were identified. In terms of internal validity, there is high confidence that the HFSE does produce improvements in interoperability (thus high confidence in the validity of the dissertation hypothesis). In terms external validity, more experimentation is needed in order to determine if there are any limitations on the classes of tools that might be integrated into the HFSE with additional effort.

IX. CONCLUSIONS

A. REVIEW OF THE DISSERTATION CONTRIBUTIONS

1. Accomplishment of the Research Goal

This dissertation provides confirming evidence of the research hypothesis. In particular, this dissertation demonstrates the following:

a. Construction of the HFSE is Feasible

As shown in Chapters III, IV, V, and VIII, it is theoretically feasible to establish a Holistic Framework for Software Engineering that consists of a Software Evolution model extended with QFD and integrated with a Federation Interoperability Object Model of subordinate software development models and tools.

b. HFSE Artifacts Can be Described Mathematically

As explained in Chapter IV it is possible to mathematically describe the HFSE constructs using graph and matrix mathematical notation and to then use linear algebra to deploy defined dependency values to other artifacts throughout a software development effort.

c. The HFSE Increases Software Tool Interoperability

Confirming evidence was presented in Chapter VIII that the application of the HFSE to a sample set of software development tools (Requisite®Pro and SEATools) increases the interoperability (data exchange and joint task execution) of the selected set.

2. Other Original and Unique Contributions

Beyond achieving the dissertation research goal, this dissertation provides several other contributions to the field of software engineering.

a. Development of a Software Development Tool Ontology Construction Methodology

Chapter III provides the blueprint for building a software development tool ontology. The methodology was adapted from other sources (notably [USCH96]), but was tailored for identifying and capturing the unique characteristics of software development tools. This methodology can be used to add additional software development tools to the already existing tool ontology presented in Chapter III.

b. Construct a Pilot Software Development Tool Ontology

Chapter III presents the beginnings of a software development tool ontology. Three separate ontologies (and their inter-relationships) are presented: a high-level software development tools ontology, an ontology that describes the Common Object Model (COM) interface of Requisite®Pro, and an ontology that describes important (from an interoperability viewpoint) classes from SEATools.

c. Use the QFD Methodology to Deploy Software Dependencies other than Quality

Chapters IV and VIII demonstrate a methodology for deploying definable software dependencies throughout a software development effort. To date, the main software dependency deployed using QFD has been a customer's view of quality as defined through the prioritization of customer requirements. While the theoretical deployment of other dependencies (such as cost, reliability, new technology, security, etc.) have been proposed by other authors, there have no proposed methods for deploying these other dependencies. This dissertation presents such a method.

d. Apply OOMI to the Software Development Tool Domain

Chapter V demonstrates how Young's Object-Oriented Model for Interoperability for heterogeneous systems was applied to an entirely different domain (other than C4I systems) by establishing a Federation Interoperability Object Model (FIOM) between software development process models and tools. This effort provided an appreciation of the difficulties in applying Young's methodology to a set of legacy heterogeneous software systems.

e. Use the HFSE to Provide Perspective Views of the Development Effort

Chapter VII explains the tool support that provides two user defined perspective views of particular aspects of a software development effort. The views presented include the Component Trace view and the Dependency Threshold view. These views allow the user to glean important decision support information from the underlying hypergraph of the software development effort. Such decision support can be later shown to provide software process and product improvements.

2. Potential Long-Term Benefits to the Field of Software Engineering

Establishing the feasibility of the HFSE and the other contributions provided in this dissertation provide significant long-term benefit to the field of software engineering. While none of these potential benefits were proven in this dissertation, the dissertation forms the foundation upon which these benefits can be proven in future research. These potential benefits can be realized in three main areas.

a. Improved Software Development Processes

The HFSE provides a framework that provides software engineers a unique holistic view of their software development processes. This holistic view should lead to identifying unrealized efficiencies and improvements in the process of developing software. Application of the HFSE should provide coherence to the development effort, ensuring that the best effort of one part of the effort feeds the best effort of the next and that each of these best efforts can be directly traced back to the features and aspects of the design that were of the greatest importance in the eyes of the customer.

b. Improved Software Products

Improvements to the software development process will inevitably lead to improvements in the product itself. Engineers will be able to more easily identify portions of the design upon which to focus additional resources and be able to identify portions of the design which would likely not be affected by a scarcity of resources. Together such efforts will lead to better software, produced on time and on budget.

c. Recognition of Unrealized Software Development Dependencies

Finally, the HFSE provides researchers a framework upon which to explore software development dependencies, potentially discovering unrealized dependencies in the process which in turn will eventually lead back to improved development methods and products.

B. RESEARCH ISSUES ADDRESSED

There were a number of issues originally identified in the dissertation proposal that were addressed while completing this research. Initially posed as questions, answers were discovered while completing the dissertation research.

1. Software QFD

a. Questions

In the application of QFD to Software, what makes software significantly different than other products? What are the implications of these differences in extending the Software Evolution model?

b. Answer

In Chapter II (particularly in the review of the work by Zultner [ZULT90, 92, 93]), there were a number of differences identified that must be accounted for when applying QFD to software. Most of this work centered on the tailoring of the QFD matrix set. This is exactly what the HFSE does; it uniquely tailors a set of QFD matrices to the actual software development process that the engineers use.

2. Automation of Requirement Prioritization

a. Question

Is it possible to "objectify" and/or "partially automate" prioritization of software requirements?

b. Answer

As discussed in Chapter IV there are a number of means of establishing dependency values (requirement prioritization being one such dependency). One of the most rigorous methodologies is Saaty's Analytical Hierarchy Process (AHP) [SAAT80]. While automated tool support is available to support these calculations (for instance ExpertChoice® software is a tool that supports AHP), inevitably human interaction is required because it has not yet been possible to formally define the numerous factors that must be considered in establishing such judgments. Thus, partial automation to support dependency valuation is feasible, but totally automating the process is beyond current capability.

3. Automation of Dependency

a. Question

In establishing the strength of dependency between Software artifacts, is it possible to "objectify" and/or "automate" the process?

b. Answer

While not confirmed in this dissertation, the automation of similar tasks in support of the dissertation indicate that it is likely possible to partially automate the establishment of dependency relationships through implementation of dependency rules during data exchange. As an example, in Chapter VIII there were numerous cases in which the “Required by” data available from SEATools (a data field that contains a text reference to which requirement led to the establishment of particular timing constraints) was used in order to manually identify the cells in the QFD correlation matrix (the Requirement x Specification matrix) that required input. It should be fairly straightforward to construct a unique rule that would automatically insert a dependency value (e.g. “3”) for any such a relationship. How to generalize such a relationship to provide a range of values would be more problematic and would require more sophisticated rule sets that could use limited supplied tool information to resolve often subtle distinctions between dependencies.

As an additional note, consideration of this research issue identified a shortcoming of the SEATools PSDL grammar, namely, the lack of a “Required By” field for establishing data streams and state streams. Currently, SEATools provides no traceability construct for the requirements leading to the creation of data streams. The PSDL grammar should be modified to add this capability.

4. QFD Dependencies

a. Question

When identifying the dependencies between QFD items, do QFD dependencies currently accommodate all the needed relationships presented by "software" or is there a need to identify new relationships?

b. Answer

While certainly not exhaustive, the work done during this dissertation research does not yet indicate the need for additional constructs beyond “dependencies” with specific attributes such as name, description, range of values, type, default value, origin, current value. However, there were issues as to how to properly interpret tool values for particular dependency values so that they could be accurately deployed. As an example, Requisite®Pro automatically tracks a dependency called “Stability.” This

requirement attribute is a statement of how frequently a particular requirement has changed when compared to other requirements. It is recorded as High, Medium, or Low. The issue to the HFSE, was how to interpret this step function during deployment. For instance, interpreting the function as High:Medium:Low = 3:2:1 produces different results than interpreting it as 9:3:1. Thus, it would be useful to conduct additional research to investigate this issue and attempt to identify characteristics that should be considered when attempting to interpret the relationship of values within a particular dependency.

5. QFD and the RH Model

a. Questions

To what objects, within the Software Evolution model, should QFD information be applied? In extending the Relational Hypergraph, is it better to encode QFD information in the step attributes or component attributes or would it be better to define new dependency objects?

b. Answer

A direct comparison study comparing the benefits of encoding QFD information in different objects within the Relational Hypergraph model was not undertaken in this dissertation. Instead, dependencies were established by superceding the `primary_input_driven` and `secondary_input_driven` dependencies of the RH model and replacing them with a range of dependency types and values. In particular, steps (activities denoting impetus to create components) were distinguished from dependencies (relationships between two components). This was done not because of any particular benefit provided by the RH model's dependencies, but instead as an effort to minimize the number of additional changes to the existing RH model.

6. Monitoring Artifacts

a. Questions

When integrating the Evolution System with the Object Federation, how will the Evolution system "monitor" activity in the object federation? How will it monitor activity in subordinate models and tools that does not trigger federation activity? Which activities should be monitored to achieve which purposes?

b. Answer

The mechanism employed in this dissertation to “monitor” activity was very rudimentary and involved the importing of static data files created by subordinate software development tools. As future research, investigation into the use of the dynamic capabilities of middleware (e.g. CORBA, COABS, etc.) should be undertaken as a means of performing active monitoring via publish and subscribe mechanisms.

7. HFSE Communications

a. Question

What medium will be used for communication between the Evolution Model and the Object Federation, between the Object Federation and subordinate tools/models (e.g. publish and subscribe, etc.)?

b. Answer

The full gamut of communications/data transport mechanisms was not explored in this dissertation. The original idea was to implement a dynamic middleware mechanism such as CORBA or COABS in order to dynamically keep track of changes to all software development artifacts. Unfortunately, this concept proved to be too ambitious and a less complex mechanism (importing static CSV files) was used to demonstrate the theoretical feasibility of the HFSE. This does not negate the need for future research that should be undertaken to prove the technical feasibility of the HFSE (as opposed to the theoretical feasibility established in this dissertation). In such a study, multiple mechanisms should be compared in order to determine the ideal communications mechanism for the HFSE.

8. HFSE and APIs

a. Questions

To be realizable in practice, is the scope of tools to be integrated with the HFSE restricted to those that provide APIs? Another issue is how to integrate GUI's – this probably cannot be done without changing them to some degree, and most real tools do not provide source code. Thus, will GUI's have to be replaced via API's?

b. Answer

In order to keep this research within a manageable scope, only two tools were examined in detail. One of these tools had a defined API (Rational's RequisitePro)

while the other tool did not (SEATools). In both cases there were no particular problems in adapting the standard output of the tools in such a way to provide meaningful input to CASES. However, this issue is far from settled. As future research, an investigation should be undertaken to determine the feasibility of integrating into the HFSE a software development tool that only provides graphic output.

9. Missing and Ambiguous Data

a. Question

How will missing and ambiguous data be accounted for in the HFSE?

b. Answer

As explained in Chapter V, Young's OOMI methodology is used to provide the bridge for missing or ambiguous data. The object relationships established in the OOMI provide a facility for software engineers to provide specific translators between separate object constructs in two or more object models. Within each of these translators, the engineer can define specific cases to handle missing or ambiguous data.

10. HFSE Extensibility

a. Questions

What attributes within the HFSE must be modified to provide extensibility? What mechanisms will be used to provide extensibility?

b. Answer

Chapter III explains the use of an underlying software development tool ontology that is leveraged to form the FIOM upon which the HFSE is based. This ontology was purposely constructed to be extensible so that other additional tools could be added in the future.

11. Process Dependencies and the HFSE

a. Questions

When analyzing the Rational Software process, do the phases of the Rational process have any effect on the visible/needed tool capabilities? What is their operational significance, if any? Do they affect the HFSE modeling considerations at all?

b. Answer

In the end, this dissertation research did not focus on the Rational Software development process. Instead the HFSE was established to be independent of

process so that any software development process could be accommodated. Chapters IV, VII, and VIII provide many examples of how any software development process could be accommodated within the framework.

12. The HFSE and GUI Consoles

a. Questions

What elements of the HFSE lend themselves to establishing a GUI (for research purposes only) that provide the researcher relevant information about the underlying interaction of the tools/models? What items of information would be of interest?

b. Answer

As discussed in Chapter VII one of the main extensions to CASES was the addition of a graphical interface for providing informational views of software development artifacts and dependencies. One of the additions included the development of a graphical project schema drawing pane in which the software development process is represented; this view shows the relationship between the major types of software development artifacts and the activities that generate them. Additionally, two user-defined views allow the software engineer to induce specific subgraphs of the underlying hypergraph of the software development effort. Together, this graphical interface provides a powerful tool for researchers to gain insight into the relationships between software development artifacts.

However, it was noted during the testing and use of CASESv2.0 that this interface could be improved. In particular, an additional useful feature would be a tool that could assist the engineer in establishing correlation values; allowing the engineer to complete pair-wise comparison of components.

C. RECOMMENDATIONS FOR FUTURE RESEARCH

The achieved short-term goal of the dissertation research demonstrated that applying the HFSE to a selected tool/model set was theoretically feasible. The long-term goals stemming from this research are to actually improve the efficiency of software development processes and to improve developed software's quality, safety, and reliability. Given the extensive scope of these long-term goals, there are a number of

items that can be identified as future research that should be undertaken after the completion of this dissertation.

1. Follow-on Hypotheses

The logical follow-on research to this dissertation is to establish that application of the HFSE actually improves the efficiency of the software development process and improves the software itself. Possible hypotheses include the following:

- Employing a Holistic Framework for Software Engineering (HFSE) improves the efficiency and effectiveness of software development processes.
- Employing a Holistic Framework for Software Engineering (HFSE) improves the quality, safety, and reliability of software.

Each of these hypotheses is at least as large an undertaking as the research conducted in this dissertation.

2. Comprehensive Model Validation

As discussed in Chapter VIII, the experiment undertaken to provide confirming evidence of the dissertation hypothesis had a number of shortcomings. An additional line of research related to the HFSE would be to undertake a more comprehensive experiment to validate the HFSE model. In order to adequately validate the HFSE or to provide evidence of any of the future hypotheses above, the researcher should undertake a "Posttest-Only Control Group Design" experiment. This experiment would adequately control both internal and external sources of invalidity.

Campbell and Stanley [CAMP63] point out that the "Posttest-Only Control Group Design" experiment is a scientifically sound method of determining the effects of an experimental variable on an observation population. This experiment can be characterized as shown in Experiment 2.

R	X	O
R		O

Experiment 2

In this experiment

R \equiv Random selection,

O \subseteq {all software development tools and models}, and

X \equiv Application of the HFSE to O.

This is an experimental design in which a randomly selected group that has experienced X is compared with a randomly selected one that has not, for the purpose of establishing the effect of X. In this specific experiment set in the context of validating the benefits of the HFSE, the groups to be compared would be the same randomly selected set of tools, randomly selected from the population of all software development tools. One group is exposed to the HFSE and one is not. Both are used to undertake identical software development efforts. Objective criteria for comparing performance of the two observation groups would be established beforehand. The experiment should be run several times to provide for a sufficiently large sample size from the available set of tools/models.

3. Additional Future Research Issues

The following issues are relevant to future research efforts:

a. HFSE Providing a Common Tool View

What is the intellectual load of having to learn the idiosyncrasies of many different, incompatible tool views? Eventually, a consistent integration layer may be needed for intellectual manageability. The purpose of this layer would be to provide simplification and standardization of the tool user view. Some re-engineering may be needed. How do these issues relate to existing work? Does this reduce the size of this overall problem?

b. Tool Replacement in the HFSE

Is it possible to replace one tool in the HFSE with another, while keeping the integrity of the HFSE? Given the increasing longevity of software development efforts, it is likely that tools used to develop the early implementation of a software system will no longer be available decades later while the software is still being maintained; what are the implications to this for the HFSE?

c. Tool Data Semantics

Is it possible to exploit the greater degree of formalization and explicit semantics of data used by software tools to go further in automatic updating than was possible for Young in the context of data in military databases?

d. Specification Tradeoff Elasticity

In software, when there is positive or negative correlation between specifications, how much trade-off in capability is possible? What are appropriate software metrics for establishing trade-offs and benchmarking? What are effective representations of "positive or negative correlations" in the context of supporting the tradeoffs? If QFD does not support these, what additional constructs are needed?

e. HFSE Data Representation

What is the best medium for representation of information in the HFSE (e.g. tree structure, hypergraph, etc.)? For what purposes is representation an issue? Is there a need to assume some common aspects for all processes? Or is this part of the meta-process (the process of managing/improving the software development process)?

f. Interoperability Tradeoffs

What is the tradeoff between interoperability via conformance to a single global data standard (e.g., VHSL for VLSI designs, step for mechanical parts, etc.) versus using multiple representations, ontologies, and translations as supported by the FIOM approach?

g. Data Standards and the HFSE

Can the needs of the HFSE be met by data standards? If not, what are the extra costs and benefits that ontologies provide? For example, type systems and inheritance rules differ from one programming language to another. Does this impact the required interoperability and integration of the models and tools? Does this issue require specific features and capabilities in the HFSE?

h. Dependency Paths and Constraints

Do artifact dependencies always follow the step-to-artifact creation path? Are there instances in which the dependencies should follow portions of the path and not follow other portions of a path? Are there underlying constraints associated with the creation of a dependency? Can a dependency be any definable metric or must it be repeatable, ordered, and able to be deconstructed? Are there any other assumptions underlying dependencies?

i. Method Tailoring

Software Development Method Tailoring [FITZ03] is becoming an important activity within software development, how can the benefits of the HFSE be applied to this activity?

j. Scalability of the HFSE Approach

How scalable is the OOMI portion of the HFSE approach? As more and more tools are added to an HFSE FIOM, does the overall federation complexity significantly degrade the usefulness of the approach? If so, are there ways to reduce this complexity (for instance, by totally reengineering the FIOM or by developing a second complementary FIOM)?

k. Sensitivity Analysis

What sensitivity analysis mechanisms can be added to the HFSE to provide useful decision-support information associated with the subjective nature of the correlation and dependency valuations?

D. CONCLUDING REMARKS

Recall that challenge of "understanding" software from a holistic perspective formed a significant part of the motivation for this dissertation research. In several U.S. Government studies performed by Overton et. al. and presented in [PARI83], Overton found that there were three main factors which reduced the rate at which a software engineer can "understand" and decipher the intent and style of software written by another: the limited rate that a person can perceive clues in a mass of software artifacts, the human tendency to require more clues than are "logically" necessary in order understand, and the human tendency to be distracted and to procrastinate. As demonstrated in this dissertation research, the HFSE provides a framework in which the "clues" are made more apparent to the software developer/maintainer.

The research presented in this dissertation should be viewed as a first step into a larger and more complex investigation related to the application of the HFSE to software development. While this dissertation demonstrated the feasibility of the approach, the larger research thread aims to actually improve software development process efficiency

and effectiveness by applying the HFSE in larger real-world contexts that will not only let software engineers work faster, but let them work smarter with greater understanding of customer desires and previous development work of the software development team. The HFSE was established by embedding the relevant portions of the Quality Function Deployment methodology into the already existing Relational Hypergraph Computer Aided Software Evolution model, then integrating this extended evolution model with a Federation Interoperability Object Model created from the tools and models use by the development team. Together, this framework provides an improved evolution-based, customer-focused holistic model upon which to develop safe, reliable software, produced on time and on budget that fully meets the Department of Defense and the nation's software requirements.

GLOSSARY

A

Analytic Hierarchy Process (AHP): a statistical process developed by [SAAT80] which allows groups of hierarchically structured entities be valued in relation to each other. Increasing, AHP is used to assist in establishing valuations in the QFD process.

Arcadia: a DARPA ISPE project in the early 1990's consisting of validated research of numerous tools that relied on using an object management system and a software process language.

atomic component: An atomic component is a component that cannot be decomposed into refined components [HARN99c].

atomic evolution step: An atomic evolution step is a step that cannot be decomposed into refined steps [HARN99c].

atomic evolutionary hypergraph: An atomic evolutionary hypergraph is an evolutionary hypergraph that cannot be decomposed into refined hypergraphs [HARN99c].

atomic SPIDER: It is an atomic step processed in different entrance relationships [HARN99c].

atomic step: An atomic step is a step that cannot be decomposed into refined steps [HARN99c].

C

cardinality: A slot facet that describes whether the slot has just one value (*single*) or more than one value (*multiple*). In Protégé-2000, Single is the default [PROT03b].

classes tab: The Protégé-2000 part used to create, view, revise, and save classes [PROT03b].

clustering: an AHP technique by which non-independent components are “clustered” into independent groups for comparison.

Component Class Representation (CCR): in the OOMI, the representation of a real world entity in a legacy system.

component management: Component management is one of CASES functions. In this function stakeholders can enter, delete, retrieve, modify, and query the attributes of atomic component from the hypertext database or software library (including software base and design database) [HARN99c].

component traceability: Component traceability is one of CASES functions. In this function an atomic component generated by its source atomic step can be traced not only by primary input which is the link between old version and new version atomic components, but also by a secondary input which is the link between source atomic step and components on which it depends, such as requirements and problem reports [HARN99c].

Component Trace View: in the HFSE, a user-defined view in which a subgraph is induced from a single atomic component. The subgraph contains only those components connected to the component of interest.

composite component: A composite component can be decomposed into refined components [HARN99c].

composite edge: A composite edge can be decomposed into refined edges in a hypergraph [HARN99c].

composite node: A composite node can be decomposed into refined nodes in a hypergraph [HARN99c].

composite step: A composite step can be decomposed into refined steps [HARN99c].

Computer-Aided Prototyping System (CAPS): CAPS is an easy to use, visual and integrated tool that can be used to rapidly design real-time applications using its PSDL editor, reusable software database, program generator, real-time scheduler, and so on [HARN99c]. CAPS has evolved into the prototyping suite known as SEATools.

Computer Aided Resuscitation Algorithm (CARA): a real-world software system used to control the amount of intravenous fluid pumped to a battlefield casualty. CARA was used as a case study for the experiment conducted in this dissertation.

Computer-Aided Software Evolution System (CASES): CASES is the automated tool support for the HFSE. It provides an evolution environment in which software development artifact dependencies are captured and tracked.

Consistency Index (CI): defined by [SAAT80] for the AHP, the CI is an intermediate calculation for measuring of the consistency a particular pair-wise comparison matrix

Consistency Ratio (CR): defined by [SAAT80] for the AHP, the CR is a measure of the consistency a particular pair-wise comparison matrix. CR's < 0.1 are considered acceptable.

constraint management: Constraint management is one of CASES functions. In this function the project organizer sets constraints that affect the scheduling of steps, such as predecessors, priorities, deadlines, estimated duration, earliest start times, finish times, as well as constraints that affect personnel assignments, like security level and skill requirements for a step [HARN99c].

Contextual Inquiry (CI): A method used by Digital Corporation for gathering customer requirements by observing customers in their work environment.

Coverage Analysis: a QFD technique for ensuring there are sufficient implementation components (e.g. are there enough specifications for implementing all of the requirements).

current component: A current component is a component a stakeholder is working on [HARN99c].

current step: A current step is a step a stakeholder is working on [HARN99c].

customer role: The roles of customers include system owners and end users [HARN99c].

D

DARPA Agent Markup Language (DAML): a DARPA sponsored project in support of the Semantic-Web in which a series of ontologies are linked together to provide semantic interoperability between domains.

dependency: The dependencies among software evolution objects are classified into four types: component-to-step, step-to-component, component-to-component, and step-to-step dependencies [HARN99c]. QFD dependencies (defined later) are a form of component-to-component dependencies.

dependency management: Dependency management is one of CASES functions. In this function the dependencies among atomic components to an atomic step can be identified and managed [HARN99c].

Dependency Threshold View: in the HFSE, a user-defined view in which a subgraph is induced from a set of atomic components. The subgraph contains only those components with dependency values greater than (or less than) a particular user-specified threshold value.

deployment, deployed: a set of calculations for a set of QFD dependency values downstream (or upstream) in a software development effort.

direct slot: A slot attached directly to a class (in contrast to a slot which is inherited) [PROT03b].

domain: A particular field of knowledge, such software engineering [PROT03b].

downstream: in the direction of temporal creation of software artifacts (e.g. code is "downstream" of requirements (in the same development cycle)).

E

end user: The end user is a person who uses the software product and manipulate the software system [HARN99c].

Evolution Control System (ECS): The ECS provides automated assistance for the software evolution process in an uncertain environment where designer tasks and their properties are always changing. An ECS has two main functions. The first is to control and manage evolving software system components (version control and configuration management). The second is to control and coordinate evolution team interactions (planning and scheduling software evolution tasks, which they refer to as evolution steps) [HARN99c].

evolution history merging: Creating a new component based on two primary input components is called software evolution history merging [HARN99c].

evolution history splitting: Creating a new component in a variant different from the original variant is called software evolution history splitting [HARN99c].

evolutionary hypergraph: An evolutionary hypergraph is a labeled, directed, and acyclic hypergraph together with component and step attributes. The evolutionary hypergraph is a multi-level structure due to the refinement of the hyperedge [HARN99c].

F

facets: The attributes of a slot. Some facets depend on the value of the type facet. For example, an integer slot type has facets for Minimum and Maximum [PROT03b].

feature model: a coherent model of the common and variable properties of concepts (and their interdependencies) of a potential software system [CZAR00].

Federation Entity (FE): the grouping of all FEVs for a single real-world entity.
Together the FE constitutes the representation of a single real-world entity between all systems in the federation.

Federation Entity View (FEV): a single relationship between a CCR and an FCR.

Federation Interoperability Object Model (FIOM): the grouping of all FEs for a federation of interest.

Federation Class Representation (FCR): the class representation of a single entity at the federation level.

forms tab: The Protégé-2000 part used to create the forms for acquiring instances of classes. It may also be used to view, revise, and save the forms [PROT03b].

G

graph model (or graph data model): The graph model represents the evolution history as a directed acyclic graph $G = [C, S, I, O]$ which is a bipartite with respect to the edges I and O . To model the hierarchical structure of the evolution history, the graph model was modified to be a graph $G = [C, S, CE, SE, I, O]$ [HARN99c].

H

Holistic Framework for Software Engineering (HFSE): a conceptual framework for establishing interoperability between software development tools as well as a methodology (with tool support) that assembles the necessary objects and interoperability constructs for tracking and leveraging the dependencies between development artifacts.

hyperedge: The hyperedge is a multi-level structure of the evolution step [HARN99c].

hypergraph: The hypergraph is a DAG (directed acyclic graph) with no looping paths [HARN99c].

hypergraph model: The hypergraph model is introduced to formalize the hierarchical structure of the evolution history in more detail [HARN99c].

I

inference rule management: Inference rule management is one of CASES functions. In this function the stakeholders can specify and adjust inference rules related to SPIDER formation, scheduling and assignment constraints, policies, special assignments, and so on, to help them resolve the design and management issues of the software development process [HARN99c].

inheritance: A parent-child (superclass-subclass) relationship between two classes. A child (subclass) inherits the slots of its parent classes (superclasses) [PROT03b].

inherited slot: A slot that is attached to a class via inheritance from a parent class [PROT03b].

input component: The input component to a current step is a set that combines a primary input component set and a secondary input component set [HARN99c].

input component search engine: The input component search engine can trace the dependencies among the software evolution components with the inference rules to find the input scope of the induced step [HARN99c].

instance (KB value): Concrete occurrence of information about a domain that is entered into a knowledge base. For example, Fran Smith might be an instance for a Name slot. An instances is entered via a form generated by Protégé-2000 [PROT03b].

instance (slot type): A type of slot whose value is the instance of a class [PROT03b].

instances tab: The Protégé-2000 part used to acquire instances of classes. It may also be used to view, revise, and save the instances [PROT03b].

Integrated Computer Aided Software Engineering (I-CASE): a software development approach that relies on integrating a several CASE tools.

Integrated Software Development Environment (ISDE): a software development approach providing common services and tools for multiple aspects of the software development effort.

Integrated Software Project (or Programming) Environment (ISPE): a software development approach providing common services and tools for multiple aspects of the software development effort. The terms ISDE and ISPE are used interchangeably.

Issue-Based Information Systems (IBIS) model: IBIS model follows the principle that the design process for complex systems is fundamentally a conversation among the stakeholders to resolve design issues. This model was extended to encompass prototype demos, analysis, and design activities and applied to design a decision support mechanism for software requirements engineering [HARN99c].

K

Kano Model: the grouping of customer requirements into three broad areas: 1) Normal requirements -- those requirements for which the customer receives proportional satisfaction upon the delivery of functionality meeting the requirements, 2) expected requirements -- those requirements for which the customer is dissatisfied if the requirements fails to be fulfilled in the delivered software, and 3) exciting requirements -- those requirements for which the customer receives positive satisfaction upon fulfillment.

knowledge-acquisition tool: A tool used to build a knowledge base by acquiring instances. In Protégé-2000, the forms comprise the KA tool [PROT03b].

knowledge base (KB): A set of instances of classes which may be used by PSMs [PROT03b].

knowledge-based system: A computer system that includes a knowledge base about a domain and programs that include rules for processing the knowledge and for solving problems relating to the domain [PROT03b].

M

minimal hypergraph: A minimal hypergraph is a minimal unit of hypergraph whose edge set has only one edge [HARN99c].

O

Object-Oriented Model for Interoperability (OOMI): a model developed by [YOUN02b] which resolves modeling differences in a federation of independently developed heterogeneous systems, thus enabling system interoperation.

OOMI Integrated Development Environment (OOMI IDE): a specialized toolset in support of the OOMI that is used to construct the FIOM.

ontology: A model of a particular field of knowledge - the concepts and their attributes, as well as the relationships between the concepts. In Protégé-2000, an ontology is represented as a set of classes with their associated slots [PROT03b].

output component: A step can generate one unique output component [HARN99c].

P

.pins file: A Protégé-2000 file in clips format that contains instances [PROT03b].

.pont file: A Protégé-2000 file in clips format that contains an ontology [PROT03b].

.pprj file: A Protégé-2000 file that contains a project. A project file contains the customized form information and references to external sources of the domain information [PROT03b].

path: A path in the hypergraph is an evolution history whose components, including nodes and hyperedges, can be traced [HARN99c].

personnel management: Personnel management is one of CASES functions. In this function project managers control the current status of the project personnel such as skill, skill level, security level, on-hand jobs, and so forth [HARN99c].

primary-input-driven hypergraph: Each path in a primary-input-driven hypergraph is constructed by primary-input-driven path [HARN99c].

primary-input-driven path: If there exist an input node and an output node to an evolutionary hyperedge that are different versions of the same component then the path from the input node via the hyperedge to the output node is called a primary-input-driven path [HARN99c].

primary input component: If there exist an input component and an output component to a step that are different versions of the same component then the input component is called a primary input component [HARN99c].

primitive component: The primitive component that is a source component can not be produced by any step [HARN99c].

project evaluation: Project evaluation is one of CASES functions. In this function after project organizers propose an evolution step as a project, this project will be evaluated by project evaluators according to the possibility analysis of executing this software evolution step [HARN99c].

project schema: a top-level evolutionary hypergraph which is an abstraction of the software development process being modeled within the HFSE. It is this abstraction which is first modeled within the CASES v2.0 drawing frame.

project team: In CASES, there are three kinds of project teams: the project organization team, the system analysis team, and the system design team [HARN99c].

Prototype System Description Language (PSDL): PSDL is a specification language that is used in CAPS. PSDL provides graphical notation for dataflow diagrams enhanced with nonprocedural control timing constraints [HARN99c].

prototyping method: The prototyping process repeats a guess/check/modify cycle until the users agree that the demonstrated behavior is acceptable [HARN99c].

Q

Quality Function Deployment (QFD): a requirements based methodology by which attributes of quality are deployed throughout a development effort.

QFD correlation: a user specified value between two atomic components that represents the strength of relationship between the components. Typical QFD correlation schemes use 0:1:3:9 to represent the strength of relationship; however, a user may specify a different scheme to meet particular needs (even one that uses negative values).

QFD dependency: a valued component attribute, which in combination with other dependency values can be “deployed” to other components.

QFD dependency deployment: a set of calculations for a set of QFD dependency values downstream (or upstream) in a software development effort.

R

Random Index (RI): in AHP [SAAT80], RI is an empirically derived average of the consistency indices (CIs) of a set of randomly generated right, diagonal, reciprocal matrices.

Rational Rose: a Rational Software Corporation software development tool that allows designers to model a software system using UML.

Rational Requisite®Pro: a Rational Software Corporation software requirements management tool.

Rational Unified Process (RUP): a software engineering process, which espouses a disciplined approach to assigning tasks and responsibilities within a software development organization. Its goal is to ensure the production of high-quality software that meets the needs of its end-users, within a predictable schedule and budget. The RUP is also considered to be a *process product*, developed and maintained by Rational® Software [RATI98].

relational hypergraph: A relational hypergraph is an evolutionary hypergraph in which the dependency relationships between components and steps can have a hierarchy of specialized interpretations [HARN99c].

Relational Hypergraph Model (RH model): The RH model is a formal model for the software evolution which can help us develop tools to manage both the activities in a software development project and the products that those activities produce [HARN99c].

relational hypergraph net: The relational hypergraph net is a relational hypergraph which transfers a primary input hypergraph and secondary input hypergraphs into a top-level evolutionary hypergraph and an atomic evolutionary hypergraph. Therefore, a relational hypergraph net includes a top-level relational hypergraph net and an atomic level relational hypergraph net [HARN99c].

reusable software evolution component: The components can be reused in software evolution processes [HARN99c].

S

secondary-input-driven hypergraph: Each path in a secondary-input-driven hypergraph is constructed by secondary-input-driven path [HARN99c].

secondary-input-driven path: If there exist an input node and an output node to an evolutionary hyperedge that are different components then the path from the input node via the hyperedge to the output node is called a secondary-input-driven path [HARN99c].

secondary input component: If there exist an input component and an output component to a step that are different components, then the input component is called a secondary input component [HARN99c].

slot: An attribute of a class. For example, a *physician class* might have *name*, *title*, and *phone number* as slots [PROT03b].

slots tab: The Protégé-2000 part that allows you to create, view, edit, and delete slots [PROT03b].

Software Development Life Cycle (SDLC): The SDLC model is called the waterfall model whose phases include requirements gathering, analysis, modeling or design, coding and testing [HARN99c].

Software Engineering Automation Tools (SEATools): the suite of software prototyping support tools developed at the Naval Postgraduate School. Evolved from CAPS and Distributed CAPS (DCAPS).

Software Engineering Body of Knowledge (SWEBOK): an ongoing IEEE project devoted to providing a "consensually-validated characterization of the bounds of the software engineering discipline" [SWEB01]. The SWEBOK *categorizes* the existing (and future) knowledge for the domain of software engineering; however, it does not attempt to *define* that knowledge.

software evolution: We consider software evolution to include all the activities that change a software system, as well as the relationships among those activities [HARN99c].

software evolution component: Software evolution components include software and all of the components that are related to software evolution, such as *criticisms, issues, requirements, specifications, modules, programs, optimizations, test scenarios, and stakeholders*, within software evolution processes [HARN99c].

software evolution history (or evolution history): We use relational hypergraph to construct software evolution history [HARN99c].

software evolution management: We use dependency rules to manage software evolution objects [HARN99c].

software evolution object: Software evolution objects include software evolution steps and software evolution components [HARN99c].

software evolution process: Software evolution process includes software prototype evolution process and software product generation process [HARN99c].

software evolution step (or evolution step): Each software evolution step has an estimated task duration, deadline, priority, and a required skill level. Software evolution steps in software evolution process include: software prototype demo step, issue analysis step, requirement analysis step, specification design step, module implementation step, program integration step, software product demo step, and software product implementation step [HARN99c].

software evolution traceability: The issues of traceability in software evolution can be represented by paths of the hypergraph [HARN99c].

software project: A software project is a project that can be built by the RH model, organized by project organizers, evaluated by project evaluators, and completed by system analysts and system designers [HARN99c].

Software Quality Function Deployment (SQFD): the QFD process applied to software.

SPIDER: SPIDER denotes the Step Processed In Different Entrance Relationships [HARN99c].

Statistical Process Control (SPC): The removal of software defects through appraisals; logging and correcting customer complaints; completing software reviews, inspections, walkthroughs; and performing software testing.

step management: Step management is one of CASES functions. In this function the content of the top-level step can be automatically generated, refined, and queried. The content of the atomic step can also be automatically generated, combined, and queried [HARN99c].

step refinement: Step refinement is one of CASES functions. In this function, the software evolution top-level step can be refined into a set of atomic steps [HARN99c].

Superfluous Artifact Analysis: an HFSE analysis by which superfluous components (artifacts in the design that are not needed and have been erroneously introduced) are identified.

T

top-level evolution step: The top-level evolution step is the root step of an evolutionary hypergraph [HARN99c].

top-level evolutionary hypergraph: The top-level evolution hypergraph is the root of an evolutionary hypergraph [HARN99c].

top-level relational hypergraph net: A top-level relational hypergraph net is composed of a set of top-level SPIDERS. The top-level relational hypergraph net describes the relationships not only among each top-level step and its input and output nodes but also among each composite node and its subnodes [HARN99c].

top-level SPIDER: This is a top-level step processed in different entrance relationships [HARN99c].

type: A slot facet that identifies the kind of values a slot may have - Any, boolean, float, instance, integer, string, or symbol [PROT03b].

U

upstream: against the direction of the temporal creation of software artifacts (e.g. requirements are "upstream" of specifications (in the same development cycle)).

V

variant: Variants represent alternative formulations of a software object with different objectives, such as running on different operating systems or serving different user communities [HARN99c].

version: A version of an object is one of the attributes of this object that can be represented as a string type containing the concatenation of an object identifier, a variant number, and a version number [HARN99c].

version control and configuration management: Version control and configuration management is one of CASES functions. In this function, a labeling function of CASES automatically determines the version and variation number of output components of a step. Software evolution process loops of CASES automatically construct the configuration management [HARN99c].

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: CASES USE CASES

A. INTRODUCTION

This appendix provides the detailed uses cases for the Computer-Aided Software Evolution System (CASES) and provides a statement as to whether the use case has been implemented in CASESv2.0. These use cases were developed in support of this dissertation in order to appropriately identify the required system responses of the evolution tool support. Figure 92 illustrates the context within which CASES operates.

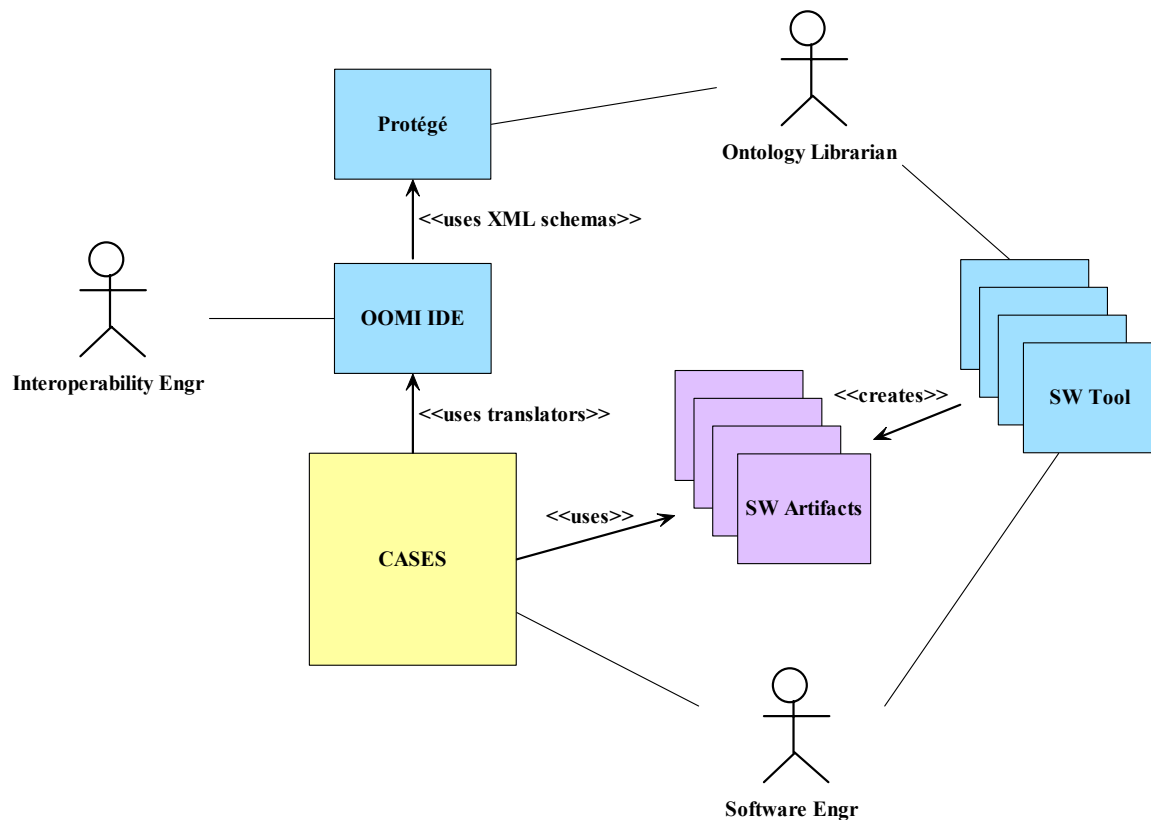


Figure 92 CASES Context Diagram

As discussed in Chapter III, the Ontology Librarian is responsible for initially developing the federation and tool ontologies and entering them into Protégé. Protégé produces the XML Schemas for the FIOM which are imported into the OOMI IDE. Note that

CASES then imports these Java translators from the OOMI IDE so that it can track software evolution artifacts of subordinate tools in the HFSE federation.

B. CASES TOP-LEVEL USE CASES

Figure 93 illustrates CASES top-level use cases. Note that the six separate use cases address the functionality required by the CASES context diagram above (Figure 92)

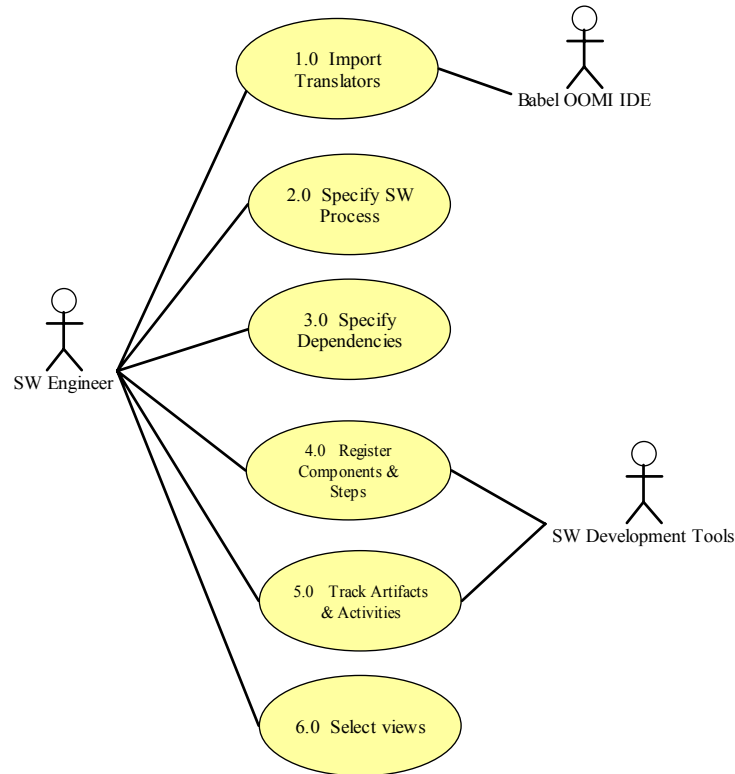


Figure 93 CASES Top-Level Use Cases

These use cases can be more fully stated as the following:

- 1.0 Import translators from Babel.
- 2.0 Software Engineer specifies Software Process to be used.
- 3.0 Software Engineer specifies component Dependencies.
- 4.0 Software Engineer registers components/steps to external tool artifacts/activities for automated tracking.
- 5.0 CASES through middleware mechanism collects/tracks artifacts and activities.
- 6.0 Software Engineer selects views of dependent artifacts.

Each of these use cases is more fully explained in the remaining sections of the appendix.

C. USE CASE 1.0: IMPORT TRANSLATORS FROM BABEL

The OOMI IDE generates Java translators that can be wrapped or embedded into middleware to translate XML documents generated by one system into XML documents that can be used by another system. This use case delineates how CASES imports those translators so that they can be used for tracking specific artifacts between the two (or more) systems. Figure 94 illustrates the use case and Table 49 provides a description of the user and system responses within the use case.

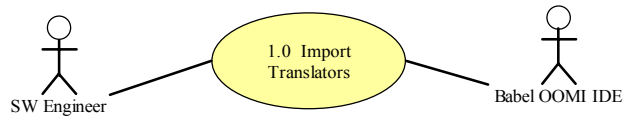


Figure 94 Use Case 1.0: Import Translators from Babel

#	Actor	System
1	The user selects "Import Translators" from the menu bar	The system responds with a file browser pointed at the current directory
2	The user browses to the desired file location containing the java translators and selects the one(s) he wishes to import.	The system loads the translators into the interoperability editor.

Table 49 Actor-System Responses for Use Case 1.0

This use case has not been implemented in CASESv2.0.

D. USE CASE 2.0: SOFTWARE ENGINEER SPECIFIES SOFTWARE PROCESS

One of the major shortcomings of previous versions of CASES was that the tool was specifically designed to work only with a single software development process model (the Evolutionary Prototyping Model). This dissertation provides an improvement over the previous versions by allowing software engineers to graphically define the software development process that they actually use, specifying the components (artifacts) that are produced in their process and the steps (activities) that they perform to create those components. Use case 2.0 is devoted to providing the engineer that functionality and is shown below in Figure 95.

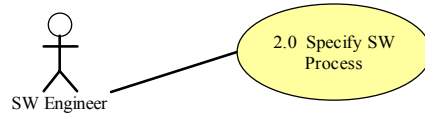


Figure 95 Use Case 2.0: Software Engineer Specifies Software Process

Because this functionality is relatively complex, this use case has been decomposed into a number of more detailed cases as follows:

- 2.1 Load existing software process.
- 2.2 Create components and component attributes.
- 2.3 Edit components and component attributes.
- 2.4 Create steps and step attributes.
- 2.5 Edit steps and step attributes.
- 2.6 Move (rearrange) components and steps.
- 2.7 Delete components and steps.
- 2.8 Decompose components into subcomponents (e.g., Specifications → SpecGroup1, SpecGroup2, etc.).
- 2.9 Decompose steps (e.g., Analyze Requirements → Establish Constraints + Formulate Questions + Formalize Requirements).

The results of this use case (and its subordinate use cases) is that CASES provides a graphic view of the software process that the engineer uses and establishes directories for the accumulation of the actual software artifacts.

1. Use Case 2.1: Load Existing Software Process (2 Scenarios)

Use case 2.1 allows a software engineer to open an already existing software process model. An existing process model could be one that the engineer had already been using (scenario 1), or it could be a "template" for the start of refining his own specific process model (scenario 2). The Actor-System sequences for these two scenarios are shown in Table 50 and Table 51 respectively.

#	Actor	System
1	The user selects "Open Existing Project Schema"	The system responds with a file browser pointed at the current directory
2	The user browses to the desired existing project schemas and selects the one he wishes to use	The system loads the project schema into the editor.
3	The user may then edit the schema as desired	The system responds as per use cases 2.2 - 2.9

Table 50 Actor-System Responses for Use Case 2.1 (Scenario 1)

#	Actor	System
1	The user selects "Open Pre-Defined Project Schema"	The system responds with a dialog that lists the available pre-defined project schemas (e.g. evolutionary prototyping model, waterfall model, spiral development model)
2	The user selects the desired schema	The system responds with a "Save As" dialog.
3	The user saves the loaded schema as a new project	The system creates new cfg, component, and step directories (all necessary project directories) in a folder with the name supplied by the user
4	The user may then edit the schema as desired	The system responds as per use cases 2.2 - 2.9

Table 51 Actor-System Responses for Use Case 2.1 (Scenario 2)

The use case defined in Scenario 1 has been implemented in CASESv2.0; the use case in Scenario 2 has not been implemented in CASESv2.0.

2. Use Case 2.2: Create Components and Component Attributes

Once CASES has been started and either a new project has been created, pre-existing project opened, or a project schema template opened; the editing frame is then available for users to create components in the schema. Actually the engineer is creating an abstract container for the components (the artifacts) of their particular software development process. Table 52 lists the Actor-System responses for creating components within CASES.

#	Actor	System
1	The user selects the "component" button from the Project Schema tool bar.	The system responds by activating the cursor to allow the user to place "components" within the drawing frame.
2	The user places a component on the drawing frame by clicking the left mouse button.	<p>The system draws a named circle on the drawing frame. Initially, the name is a unique default component identifier.</p> <p>The system resets the cursor to allow the placement of additional components.</p>

Table 52 Actor-System Responses for Use Case 2.2

This use case (use case 2.2) has been implemented in CASESv2.0.

3. Use Case 2.3: Edit Components and Component Attributes (2 Scenarios)

After the user has created components within the project schema drawing frame, he can then edit the attributes of those components through two different means illustrated in the Actor-System responses shown in Table 53 and Table 54.

#	Actor	System
1	After selecting the "select" button on the Project Schema tool bar, the user double clicks on an established component	The system responds by displaying the component attribute dialog.
2	<p>The user edits the attribute fields in the dialog:</p> <ul style="list-style-type: none"> • Component ID: a unique component identifier • Component Name: a 50 character string • Component Description: a 250 character text field 	<p>The system reports an error (and prevents the user from exiting the dialog) if the user attempts to create a component with an ID that is identical to an existing component</p> <p>The system stores the component attributes.</p> <p>The system renames the component file in accordance with the component ID.</p>

Table 53 Actor-System Responses for Use Case 2.3 (Scenario 1)

#	Actor	System
1	After selecting the "select" button on the Project Schema tool bar, the user <u>right</u> clicks on an established component.	The system offers the user an extended menu of the following: <ul style="list-style-type: none"> • Properties • Decompose • Delete • QFD • Import CSV File
2	The user selects "Properties"	The system responds by displaying the component attribute dialog.
3	The user edits the attribute fields in the dialog. <ul style="list-style-type: none"> • Component ID: a unique component identifier • Component Name: a 50 character string • Component Description: a 250 character text field 	<p>The system reports an error (and prevents the user from exiting the dialog) if the user attempts to create a component with an ID that is identical to an existing component</p> <p>The system stores the component attributes.</p> <p>The system renames the component file in accordance with the component ID.</p>

Table 54 Actor-System Responses for Use Case 2.3 (Scenario 2)

Both scenarios of use case 2.3 have been implemented in CASESv2.0.

4. Use Case 2.4: Create Steps and Step Attributes

In much the same way that the user created the components, the user can create steps for the project schema. The Actor-System responses for the use case are shown in Table 55.

#	Actor	System
1	The user selects the "step" button from the Project Schema tool bar.	The system responds by activating the cursor to allow the user to draw "steps" between exiting components within the drawing frame.
2	The user draws a step by left clicking once on an existing component and then left clicking a second time on another component.	<p>The system draws a named line on the drawing frame between the two components. Initially, the name is a unique default step identifier.</p> <p>The system resets the cursor to allow the placement of additional steps.</p>

Table 55 Actor-System Responses for Use Case 2.4

This use case (use case 2.4) has been implemented in CASESv2.0.

5. Use Case 2.5: Edit Steps and Step Attributes (2 Scenarios)

After the user has created steps within the project schema drawing frame, he can then edit the attributes of those steps through two different means illustrated in the Actor-System responses shown in Table 56 and Table 57.

#	Actor	System
1	After selecting the "select" button on the Project Schema tool bar, the user double clicks on an established step.	The system responds by displaying the step attribute dialog.
2	<p>The user edits the attribute fields in the dialog.</p> <ul style="list-style-type: none"> • Step ID: a unique step identifier • Step Name: a 50 character string • Step Description: a 250 character text field 	<p>The system reports an error (and prevents the user from exiting the dialog) if the user attempts to create a step with an ID that is identical to an existing step.</p> <p>The system stores the step attributes.</p> <p>The system renames the step file in accordance with the step ID.</p>

Table 56 Actor-System Responses for Use Case 2.5 (Scenario 1)

#	Actor	System
1	After selecting the "select" button on the Project Schema tool bar, the user <u>right</u> clicks on an established step.	The system offers the user an extended menu as follows: <ul style="list-style-type: none"> • Properties • Decompose • Delete • QFD • Import CSV File
2	The user selects "Properties"	The system responds by displaying the step attribute dialog.
3	The user edits the attribute fields in the dialog. <ul style="list-style-type: none"> • Step ID: a unique step identifier • Step Name: a 50 character string • Step Description: a 250 character text field 	The system reports an error (and prevents the user from exiting the dialog) if the user attempts to create a step with an ID that is identical to an existing step. The system stores the step attributes. The system renames the step file in accordance with the step ID.

Table 57 Actor-System Responses for Use Case 2.5 (Scenario 2)

Both scenarios of use case 2.5 have been implemented in CASESv2.0

6. Use Case 2.6: Move (Rearrange) Components

Once placed on the drawing frame, the user can move components around within the project schema. Table 58 describes the Actor-System responses for use case 2.6.

#	Actor	System
1	After selecting the "select" button on the Project Schema tool bar, the user single clicks and holds onto an established component. The user drags the component to a new desired location.	The system responds by visually cueing the user that the component has been selected (e.g. outlines the component in bold). Then the system drags the component to the new location designated by the user. Any step lines attached to the "moved" component will also move along with the component.

Table 58 Actor-System Responses for Use Case 2.6

With the exception of providing a visual cue (bold outline), this use case (use case 2.6) has been implemented in CASESv2.0.

7. Use Case 2.7: Delete Components and Steps (2 Scenarios)

The user can delete components and steps by two different means as described by the Actor-System responses in Table 59 and Table 60 below.

#	Actor	System
1	After selecting the "select" button on the Project Schema tool bar, the user <u>right</u> clicks on an established component or step.	The system offers the user an extended menu as follows: <ul style="list-style-type: none">• Properties• Decompose• Delete• QFD• Import CSV File
2	The user selects "Delete"	The system responds with a "Do you really want to Delete?" dialog.
3	The user selects either "Delete" or "Cancel"	If the user selects "Delete", the component (and any steps connected to the component) or step is deleted. The system deletes all appropriate files. If the user selects "Cancel", then the system closes the dialog without deleting the item.

Table 59 Actor-System Responses for Use Case 2.7 (Scenario 1)

#	Actor	System
1	After selecting the "select" button on the Project Schema tool bar, the user selects an established component or step with a single left mouse click.	The system responds by visually queuing the user that the component has been selected (e.g. outlines the component in bold).
2	The user presses the "Delete" button on the keyboard.	The system responds with a "Do you really want to Delete?" dialog.
3	The user selects either "Delete" or "Cancel" in the dialog.	<p>If the user selects "Delete", the component (and any steps connected to the component) or step is deleted.</p> <p>The system deletes all appropriate files.</p> <p>If the user selects "Cancel", then the system closes the dialog without deleting the item.</p>

Table 60 Actor-System Responses for Use Case 2.7 (Scenario 2)

Neither of the scenarios of use case 2.7 has been implemented in CASESv2.0. It is currently not possible to delete components; instead, the user must reconstruct the project schema.

8. Use Case 2.8: Decompose Components into Subcomponents

After the user has placed components on the drawing frame, he can then decompose a particular component into subcomponents (e.g., Specifications → Specification Group1, Specification Group2, etc.). Actor-System responses for this use case are described in Table 61.


#	Actor	System
1	After selecting the "select" button on the Project Schema tool bar, the user <u>right</u> clicks on an established component.	The system offers the user an extended menu as follows: <ul style="list-style-type: none"> • Properties • Decompose • Delete • QFD • Import CSV File
2	The user selects "Decompose"	The system responds with a new drawing frame. Existing input and output steps from the "parent" component are automatically placed into the new drawing frame with an "External" label at the appropriate end.
3	The user continues to define the software development process in accordance with use cases 2.2 - 2.9	The system updates the "parent" component attributes. The system provides a visual queue that the component is decomposable (double circle). e.g. 

Table 61 Actor-System Responses for Use Case 2.8

Use case 2.8 has not been implemented in CASESv2.0.

9. Use Case 2.9: Decompose Steps

The user has the ability to decompose steps into a sequence of atomic steps and components (e.g., “Analyze Requirements” becomes (Establish Constraints → Constraints → Formulate Questions → Questions → Formalize Requirements→ Formal Requirements)). The Actor-System responses are described in Table 62.

#	Actor	System
1	After selecting the "select" button on the Project Schema tool bar, the user <u>right</u> clicks on an established step.	The system offers the user an extended menu as follows: <ul style="list-style-type: none"> • Properties • Decompose • Delete • QFD • Import CSV File
2	The user selects "Decompose"	The system responds with a new drawing frame. Existing input and output components from the "parent" step are automatically placed into the new drawing frame.
3	The user defines the additional more detailed description of the step of the software development process in accordance with use cases 2.4 - 2.7 and 2.9 by adding steps and components.	The system updates the "parent" step attributes. The system provides a visual queue that the step is decomposable (double line). Dev Spec e.g. <u><u> </u></u>

Table 62 Actor-System Responses for Use Case 2.9

Use case 2.9 has not been implemented in CASESv2.0.

E. USE CASE 3.0: SOFTWARE ENGINEER SPECIFIES COMPONENT DEPENDENCIES

After the software engineer has constructed the project schema that lays out the software development process that he is using, the next step is to specify those dependencies that he wishes to "deploy" throughout the development effort. This use case is shown below in Figure 96.

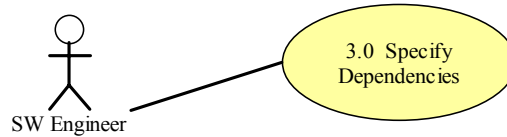


Figure 96 Use Case 3.0: Software Engineer Specifies Component Dependencies

This use case is then further decomposed into several more detailed use cases as follows:

- 3.1 Create dependency (establish type of dependency and dependency attributes).
- 3.2 Establish dependency linkages between components.
- 3.3 Deploy dependency through the development effort.

The result of this set of use cases is that the system provides the user the ability to define a particular dependency. The user can then edit the relationships between components and set specific values for the dependency related to specific components within a QFD matrix structure. The user can then "deploy" the dependency to the next component by clicking the "calculate" button or throughout the entire development effort by clicking the "synchronize" button.

1. Use Case 3.1: Create Dependency

In this use case the user establishes type of dependency and defines dependency attributes. The Actor-System responses are described below in Table 63

#	Actor	System
1	The user selects the "Create Dependency" menu item.	The system displays the "New Dependency" dialog.
2	The user inputs attributes for the new dependency as follows: <ul style="list-style-type: none"> • Short Name • Description • Dependency Type (e.g. Risk, Safety, Parent/child, etc.) • Dependency Value Range (if applicable -- Boolean, 1-9, +/-, 1/3/9, based on QFD Methodology) • Dependency Default Value • The origin component of the dependency 	The system stores the attributes.

Table 63 Actor-System Responses for Use Case 3.1

Only a portion of this use case (use case 3.1) has been implemented in CASESv2.0. The “Dependency Type” and “Dependency Value Range” currently have no functionality in the system. These entries exist in the creation dialog, but currently serve no purpose.

2. Use Case 3.2: Establish Dependency Linkages Between Components (2 Scenarios)

After creating a dependency, the user can then edit the dependency values and the correlation between components that the dependency is to be "deployed across." There are two scenarios for establishing the correlations. In Scenario 1, the engineer defines dependencies between different artifacts (e.g. R1 → S3). In scenario 2 the user establishes dependencies between the same type of component (e.g., R1 → R3.2). The Actor-System responses for these two scenarios are described in Table 64 and Table 65 respectively.

#	Actor	System																														
1	After selecting the "select" button on the Project Schema tool bar, the user <u>right</u> clicks on an established step.	<p>The system offers the user an extended menu as follows:</p> <ul style="list-style-type: none">• Properties• Decompose• Delete• QFD• Import CSV File																														
2	The user selects "QFD" and further selects the particular dependency that he wants to edit.	<p>The system produces a QFD matrix rectangular matrix.</p> <p>Inside the matrix are correlations between the two components. The value of the dependency is the default value specified in the create dependency dialog or an imported value. The initial correlation value in the matrix is set to "0". A color (e.g. yellow) is used to indicate which values the user has not yet edited as follows:</p> <table><tr><td></td><td></td><td>Spec1</td><td>Spec2</td><td>Spec3</td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td>Req1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>Req2</td><td>4</td><td>0</td><td>3</td><td>0</td></tr><tr><td>Req3</td><td>3</td><td>0</td><td>0</td><td>0</td></tr><tr><td>Req4</td><td>4</td><td>0</td><td>1</td><td>0</td></tr></table>			Spec1	Spec2	Spec3						Req1	1	0	0	0	Req2	4	0	3	0	Req3	3	0	0	0	Req4	4	0	1	0
		Spec1	Spec2	Spec3																												
Req1	1	0	0	0																												
Req2	4	0	3	0																												
Req3	3	0	0	0																												
Req4	4	0	1	0																												
3	The user edits values of dependency and correlation	The system records the values and saves them to file.																														

Table 64 Actor-System Responses for Use Case 3.2 (Scenario 1)

#	Actor	System																									
1	After selecting the "select" button on the Project Schema tool bar, the user <u>right</u> clicks on an established component.	<p>The system offers the user an extended menu as follows:</p> <ul style="list-style-type: none">• Properties• Decompose• Delete• QFD• Import CSV File																									
2	The user selects "QFD" and further selects the particular dependency that he wants to edit.	<p>The system produces a QFD matrix rectangular matrix.</p> <p>Inside the matrix are correlations between the same components. The default value of the dependency is the default value specified in the create dependency dialog. The initial correlation value in the matrix is set to "0". A color (e.g. yellow) is used to indicate which values the user has not yet edited. A different color (e.g. red) is used to indicate un-needed correlations between the same set of components as follows:</p> <table><tr><td></td><td></td><td>Req1</td><td>Req2</td><td>Req3</td></tr><tr><td></td><td></td><td>1</td><td>4</td><td>3</td></tr><tr><td>Req1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>Req2</td><td>4</td><td>0</td><td>0</td><td>0</td></tr><tr><td>Req3</td><td>3</td><td>0</td><td>0</td><td>0</td></tr></table>			Req1	Req2	Req3			1	4	3	Req1	1	0	0	0	Req2	4	0	0	0	Req3	3	0	0	0
		Req1	Req2	Req3																							
		1	4	3																							
Req1	1	0	0	0																							
Req2	4	0	0	0																							
Req3	3	0	0	0																							
3	The user edits values of dependency and correlation.	The system records the values and saves them to file.																									

Table 65 Actor-System Responses for Use Case 3.2 (Scenario 2)

Except for the use of color-coding to cue the user as to which entries have been edited, use case 3.2 Scenario 1 has been implemented in CASESv2.0. Use Case 3.2 Scenario 2 has been implemented; however, this scenario currently serves no function. Eventually, Scenario 2 will be used to establish trade-offs between the same type of components.

3. Use Case 3.3: Deploy Dependency Through the Development Effort (2 Scenarios)

After creating a dependency, correlation values, and dependency values, the user can deploy the dependency across components. There are two scenarios for deploying the dependency shown in Table 66 and Table 67.

#	Actor	System
1	After the user has edited all desired values of dependency and correlation, he selects the "Calculate" button on the QFD dialog.	The system automatically calculates the appropriate value of dependency for the deployed component and displays those values in the QFD dialog

Table 66 Actor-System Responses for Use Case 3.3 (Scenario 1)

#	Actor	System
1	After the user has edited all desired values of dependency and correlation, he selects the "Synchronize" button on the tool bar.	The system automatically calculates the appropriate values of dependency for all QFD matrices in the system. The appropriate values of the dependency are displayed the next time the user opens a particular QFD dialog.

Table 67 Actor-System Responses for Use Case 3.3 (Scenario 2)

Both scenarios of Use Case 3.3 have been implemented in CASESv2.0.

F. REGISTER COMPONENTS AND STEPS TO EXTERNAL TOOL ARTIFACTS AND ACTIVITIES

The project schema created in use case 1.0 is an abstract representation of artifacts created in the software development effort. Use Case 4.0 allows CASES to import data about real artifacts in other software tools so that dependencies between these artifacts can be established and tracked. Figure 97 below illustrates this use case.



Figure 97 Use Case 4.0: Register Components and Steps to External Tool Artifacts and Activities

This use case is further decomposed into three subordinate use cases as follows:

- 4.1 Map Components to Tool Objects.
- 4.2 Map Steps to Tool Activities/Methods.
- 4.3 Insert Translators.

The result of this set of use cases is that CASES shows a graphic view of the components and the tool objects to which they correspond. It also shows a graphic view of the steps and the tool activities and methods to which they correspond and embeds OOMI translators into middleware so that appropriate interoperability translation can take place.

1. Use Case 4.1: Map Components to Tool Objects

In this use case the user imports a Comma Separated Value (CSV) file into CASES. This CSV file contains lists of data associated with real artifacts produced in other tools. The Actor-System response for this use case is described in Table 68.

#	Actor	System
1	After selecting the "select" button on the Project Schema tool bar, the user <u>right</u> clicks on an established step.	The system offers the user an extended menu as follows: <ul style="list-style-type: none"> • Properties • Decompose • Delete • QFD • Import CSV File
2	The user selects "Import CSV File"	The system responds with a file browser pointed at the current directory
3	The user browses to the desired CSV file and selects the one he wishes to map against that particular set of components	The system loads the CSV file into component directory.

Table 68 Actor-System Responses for Use Case 4.1

Use Case 4.1 has been implemented in CASESv2.0. However, once Use Cases 4.2 and 4.3 are implemented (in future versions of CASES), this use case should be re-written to take advantage of the middleware mechanism being used.

2. Use Case 4.2: Map Steps to Tool Activities/Methods

In this use case, the user identifies actual method calls between tools so that as artifacts are created, CASES can monitor that creation and automatically and dynamically update the list and values of components (artifacts). This use case has not been implemented in CASESv2.0.

3. Use Case 4.3: Insert Translators

In this use case, the user automatically inserts the OOMI IDE supplied translators into the CASES middleware ORB and registers them against external tool objects identified in other tool ORBs. This use case has not been implemented in CASESv2.0.

G. USE CASE 5.0: CASES THROUGH MIDDLEWARE MECHANISM COLLECTS/TRACKS ARTIFACTS AND ACTIVITIES

CASES monitors and tracks the creation, modification, and deletion of artifacts throughout the software development life cycle. Artifacts and dependencies are automatically updated. Figure 98 illustrates this use case and Table 69 describes the Actor-System responses.



Figure 98 Use Case 5.0: Register

#	Actor	System
1	Through CASES, user launches Software Development tool (e.g. Word, ReqPro, CAPS, etc.)	The tool opens.
2	The user uses the tool.	<p>The tool (through the middleware) publishes changes to created artifacts.</p> <p>The system (CASES) tracks the changes to created artifacts.</p> <p>The system (CASES) translates appropriate artifacts and passes them to other launched tools.</p>

Table 69 Actor-System Responses for Use Case 5.0

This use case, although defined, is not implemented in CASESv2.0. As of now, all tracking is done through the importing of static data files (i.e. .csv files). The files can be created as often as a user desires, but the process is not yet dynamic.

H. USE CASE 6.0: SELECT VIEWS OF ARTIFACT DEPENDENCIES

In this particular use case the software engineer can define particular views of artifact dependencies. In essence the engineer is taking a "slice" of the underlying hypergraph representation of the software development effort. This slice is tailored to the specific set of artifacts and dependencies that the user specifies. Figure 99 below illustrates the use case.

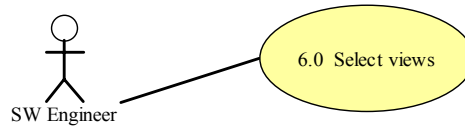


Figure 99 Use Case 6.0: Select Views of Artifact Dependencies

This use case is further decomposed into subordinate use cases as follows:

- 6.1 Software Engineer selects a view based on single dependency link (e.g. $R \rightarrow S$, thus you get the house or roof of the QFD matrix).
- 6.2 Software Engineer selects a view based on single component (e.g. $S1.2 \rightarrow$ entire induced subgraph backed out from S1.2).
- 6.3 Software Engineer selects a view based on a threshold dependency value for a particular type of dependency (e.g. Risk threshold \geq mean + 1 standard deviation \rightarrow entire induced subgraph where risk $\geq \mu + \sigma$)

The results of this use case will provide the software engineer three different means of obtaining decision support information about the software development effort

1. Software Engineer Selects a View Based on Single Dependency Link (2 Scenarios)

In this use case the engineer can view the QFD matrix (either "house" or "roof") for any set of components. Table 70 describes the Actor-System responses for viewing the QFD "house" for two different types of components and Table 71 describes the Actor-System responses for viewing the QFD "roof" for the same type of components.

#	Actor	System																														
1	After selecting the "select" button on the Project Schema tool bar, the user <u>right</u> clicks on an established step.	<p>The system offers the user an extended menu as follows:</p> <ul style="list-style-type: none">• Properties• Decompose• Delete• QFD• Import CSV File																														
2	The user selects "QFD" and further selects the particular dependency that he wants to view.	<p>The system produces a QFD matrix rectangular matrix.</p> <p>Inside the matrix are correlations between the two components and values of the dependencies as follows:</p> <table><tr><td></td><td></td><td>Spec1</td><td>Spec2</td><td>Spec3</td></tr><tr><td></td><td></td><td>1.57</td><td>5.52</td><td>2.89</td></tr><tr><td>Req1</td><td>1</td><td>0</td><td>0</td><td>9</td></tr><tr><td>Req2</td><td>3</td><td>0</td><td>3</td><td>0</td></tr><tr><td>Req3</td><td>4</td><td>0</td><td>3</td><td>0</td></tr><tr><td>Req4</td><td>2</td><td>3</td><td>0</td><td>1</td></tr></table>			Spec1	Spec2	Spec3			1.57	5.52	2.89	Req1	1	0	0	9	Req2	3	0	3	0	Req3	4	0	3	0	Req4	2	3	0	1
		Spec1	Spec2	Spec3																												
		1.57	5.52	2.89																												
Req1	1	0	0	9																												
Req2	3	0	3	0																												
Req3	4	0	3	0																												
Req4	2	3	0	1																												

Table 70 Actor-System Responses for Use Case 6.1 (Scenario 1)

#	Actor	System																									
1	After selecting the "select" button on the Project Schema tool bar, the user <u>right</u> clicks on an established component.	<p>The system offers the user an extended menu as follows:</p> <ul style="list-style-type: none">• Properties• Decompose• Delete• QFD• Import CSV File																									
2	The user selects "QFD" and further selects the particular dependency that he wants to view.	<p>The system produces a QFD matrix rectangular matrix.</p> <p>Inside the matrix are correlations between the same components. The value of the dependency is displayed. A color (e.g. red) is used to indicate un-needed correlations between the same set of components as follows:</p> <table><tr><td></td><td></td><td>Req1</td><td>Req2</td><td>Req3</td></tr><tr><td></td><td></td><td>1</td><td>4</td><td>3</td></tr><tr><td>Req1</td><td>1</td><td>0</td><td>1</td><td>3</td></tr><tr><td>Req2</td><td>4</td><td>0</td><td>0</td><td>0</td></tr><tr><td>Req3</td><td>3</td><td>0</td><td>0</td><td>0</td></tr></table>			Req1	Req2	Req3			1	4	3	Req1	1	0	1	3	Req2	4	0	0	0	Req3	3	0	0	0
		Req1	Req2	Req3																							
		1	4	3																							
Req1	1	0	1	3																							
Req2	4	0	0	0																							
Req3	3	0	0	0																							

Table 71 Actor-System Responses for Use Case 6.1 (Scenario 2)

Both scenarios of Use Case 6.1 have been implemented in CASESv2.0.

2. Use Case 6.2: Software Engineer Selects a View Based on Single Component

In this use case the user identifies a particular component and threshold correlation value (e.g. S1.2 threshold 2 → entire induced subgraph backed out from S1.2 where correlations ≥ 2). The system then identifies and displays all connected components with correlation values greater than that threshold. Table 72 describes the Actor-System responses.

#	Actor	System
1	With a QFD matrix open, the user selects "Trace" from the dialog "View" menu.	The system responds by providing a "Trace" dialog
2	The user selects the component to be traced from and the threshold correlation value	<p>The system responds by providing "culled" QFD matrices related to each step that show only those connected components with correlation values greater than the threshold.</p> <p>These components represent a "slice" of the underlying hypergraph that are connected to the selected component with a correlation greater than the selected threshold.</p>

Table 72 Actor-System Responses for Use Case 6.2

Use Case 6.2 has been implemented in CASESv2.0.

3. Use Case 6.3: Software Engineer Selects a View Based on a Threshold Dependency Value for a Particular Type of Dependency

In this use case the user identifies a particular type of dependency and a threshold value. The system then displays all components with dependency values greater or equal to than (or less than or equal to) that threshold value (e.g. Risk threshold $\geq \text{mean} + 1$ standard deviation \rightarrow entire induced subgraph where risk $\geq \mu + \sigma$). Table 73 describes the Actor-System responses for this use case.

#	Actor	System
1	With a QFD matrix open to a particular type of dependency, the user selects "User-Defined" from the dialog "View" menu.	The system responds by providing a "User-Defined" dialog
2	The user selects the amount of the standard deviation (from the mean) and whether he is interested in the values "greater than or equal to" or "less than or equal to" that threshold	<p>The system responds by providing a "culled" QFD matrix that shows only those components with a dependency value that satisfies the threshold criteria.</p> <p>These components represent a "slice" of the underlying hypergraph that meet the user's threshold criteria.</p>

Table 73 Actor-System Responses for Use Case 6.3

Use Case 6.3 has been implemented in CASESv2.0.

APPENDIX B: CARA INFUSION PUMP REQUIREMENTS FROM REQUISITE PRO

A. INTRODUCTION

Section B of this appendix provides a listing of the requirements of the CARA Infusion Pump [WRAI02a] specified within Requisite®Pro. The requirement numbers and descriptions (Feature Tag and Requirement Text) come directly from [WRAI02c]¹; the requirement names were derived from the general description of the requirement. The values for AHP Priority, Requirements Clarity, and Safety were artificially derived and created as user defined Requisite®Pro requirements attributes in order to provide meaningful data for the software experiment delineated in this dissertation. “AHP Priority” values are requirement priority values calculated using the Analytic Hierarchy Process [SAAT80] and then multiplied by 500 (AHP values typically sum to 1). Requirements clarity values are integer values based on the number and applicability of questions posed by developers about individual requirements; the higher the value, the less clear is the requirement (i.e. a high value indicates that a number of clarifying questions had to be asked about a particular requirement). Safety values are assigned on a 1 to 5 scale with 1 being low safety impact to the design and 5 being high safety impact to the design.

B. CARA REQUIREMENTS SPECIFIED IN REQUISITE®PRO

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT1	CARA On	The CARA will be operational whenever the LSTAT is powered on.	1.169	3	1

¹ Reprinted with permission of the Walter Reed Army Institute of Research, Dr. Stephen A. Van Albert, May 2003.

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT2	Pump Status Display	The display will show a message indicating the current status of the pump. One of the following indications will appear:	1.403	0	2
FEAT2.1	Not Plugged In	The pump is not plugged in	1.123	0	2
FEAT2.2	Plugged In Status Unknown	The pump is plugged in and its operational status is unknown	1.123	0	2
FEAT2.3	Plugged In Manual Mode	The pump is plugged in and is running in manual mode	1.123	0	2
FEAT2.4	Plugged In but Stopped	The pump is plugged in and is stopped	1.123	0	2
FEAT2.5	Plugged In Auto-Control Mode	The pump is plugged in and is operating in auto-control mode	1.123	0	2
FEAT3	Monitor Pump	CARA will monitor the pump connector on the LSTAT to determine when a pump is plugged in	7.016	9	2
FEAT4	Pump Detected Msg Display	When the pump is detected an appropriate message will be displayed.	7.016	0	2
FEAT5	Pump Detected Timestamp	When the pump is detected a timestamp will be entered into a resuscitation file.	7.016	0	3
FEAT6	Continuity Check	Upon connection CARA will continuously check continuity on all wires going to the pump.	1.403	13	2
FEAT6.1	Discontinuity Message Display	If a discontinuity is detected on any lines, CARA will display appropriate messages.	1.403	7	2
FEAT6.2	Discontinuity Alarm	If a discontinuity is detected on any lines, CARA will issue appropriate alarms.	1.754	7	4
FEAT6.3	Discontinuity Terminate AC	If a discontinuity is detected while in auto-control, CARA will terminate auto-control	2.456	16	5
FEAT7	Occlusion Line Monitoring	The CARA will monitor the occlusion lines whenever the pump is plugged in.	1.052	0	3

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT7.1	Occlusion Detected	If an occlusion fault is detected	1.052	0	3
FEAT7.1.1	Occlusion Display Msg	An appropriate error message should is issued.	1.052	0	2
FEAT7.1.2	Occlusion Level 1 Alarm	A level 1 alarm should is issued	1.754	3	4
FEAT7.1.3	Occlusion Terminate AC	If an occlusion is detected while in auto-control, CARA will terminate auto-control	2.105	9	5
FEAT8	Air OK Monitoring	The CARA will monitor the Air OK line whenever the pump is plugged in.	1.403	12	2
FEAT8.1	Air OK Remains Low	If the Air OK signal remains low for 10 seconds	1.052	4	3
FEAT8.1.1	AirOK Msg Display	An appropriate error message should is issued.	1.052	4	2
FEAT8.1.2	AirOK Level 1 Alarm	A level 1 alarm should is issued	1.403	7	4
FEAT8.1.3	AirOK Terminate AC	If an air fault is detected while in auto-control, CARA will terminate auto-control	2.105	13	5
FEAT9	EMF Monitoring	CARA should monitor the back EMF line from the pump to keep track of infused fluids by polling immediately when the pump is plugged in and then on every even 5-second clock interval while the pump remains plugged in.	2.339	19	2
FEAT10	Back EMF Detected in Manual Mode	If back EMF is detected in manual mode the volume infused should be calculated (Manual mode is defined as the time when the pump is connected to the LSTAT and is infusing fluid into a patient using the hardware flow setting on the pump.)	2.806	0	2

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT10.1	Log Cumulative Volume	The cumulative volume infused from the time of CARA initialization should be logged into the resuscitation file with the time every minute on the minute while the pump is plugged in.	2.806	24	3
FEAT10.2	Graph Trend Cumulative Volume	The volume infused should be trended on a graphical display every minute	1.403	9	2
FEAT11	Calculate Flow Rate	The flow rate should be calculated from the back EMF on every	1.169	9	2
FEAT11.1	Display Flow Rate	The flow rate should be displayed continuously with the display being updated with every new reading.	2.689	3	2
FEAT11.1.1	EMF Unobtained Flow Rate Unknown	If the EMF reading cannot be obtained, the display should indicate that the flow rate is unknown.	2.222	12	2
FEAT11.2	Flow Rate Logged	The average flow rate for the past minute should be written to the resuscitation file once per minute on the minute while the pump is plugged in.	2.339	13	3
FEAT11.2.1	Unknown Flow Rate Logged	If the EMF reading cannot be obtained the log entry should indicate that the average flow rate is unknown for the minute.	1.637	16	3
FEAT11.2.2	EMF Unobtained or Zero, Terminate AC	If the EMF reading cannot be obtained or is zero while in autocontrol, CARA will terminate auto-control	1.637	16	5
FEAT12	EMF Present, Log Manual Mode	If back EMF is present, the fact that the pump is in manual mode should also be recorded in the resuscitation file	2.806	0	3
FEAT12.1	Display Manual Mode	The fact that the pump is in manual mode should also be shown on the display.	1.871	0	2

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT13	BPs Various Sources	The CARA should be able to use a blood pressure from various sources as the input into the CARA algorithm. Blood pressure sources (arterial line, cuff, other noninvasive pressures [pulse wave transmission, etc.]) will be prioritized based on quality.	1.169	3	3
FEAT13.1	Corroborated A-line Priority 1	A corroborated A-line is use priority 1	4.677	9	3
FEAT13.2	Corroborated Pulse Wave Priority 2	A corroborated pulse wave pressure is use priority 2	3.508	9	2
FEAT13.3	Cuff Pressure Priority 3	A cuff pressure is use priority 3	2.339	9	1
FEAT14	Manual Model Log BP	If the CARA detects a blood pressure during manual mode, it should log the pressure (both systolic and diastolic values) to the resuscitation file every minute on the minute, coincident with any other logging that is occurring on the minute.	2.339	0	2
FEAT14.1	Avg BP Logged	If the source reports the blood pressure more than once a minute, an average over the minute should be stored into the resuscitation file.	2.572	0	2
FEAT14.1.1	BP Source Logged	Along with the blood pressure, the source of the BP should be recorded into the resuscitation file	1.871	0	2
FEAT14.1.2	BP Time Logged	The time of the BP should be recorded into the resuscitation file.	2.572	0	2
FEAT14.1.3	BP Display w/ Infused Volume	The blood pressure that is stored to file should also be graphed on the same display as the infused volume	1.169	9	2

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT14.1.4	BP Value on Display	The numerical value of the blood pressure should be displayed as well.	1.169	0	2
FEAT15	Respond to Lost BP Sources	During resuscitation CARA should respond to any lost blood pressure sources	2.105	27	4
FEAT15.1	Display Msg, Lost BP Source	With an appropriate message	3.508	3	2
FEAT15.2	Level 1 Alarm, Lost BP Source	With a level 1 alarm	4.911	18	4
FEAT15.3	Logged, Lost BP Source	With a notation in the resuscitation file	3.508	3	3
FEAT16	CARA Ready for AC	When CARA determines that 1) a pump is plugged in and not stopped, 2) an infusate with an impedance within tolerance is in place, and 3) the occlusion line is clear,	0.327	0	2
FEAT16.1	Display AC Ready	The display will show a "CARA Status OK" message indicating that CARA is ready to start auto-control and a Start Auto-control button.	1.473	0	2
FEAT16.2	Display Mean BP	The display will also show the default mean blood pressure of 70 mmHg to which CARA will titrate.	1.473	0	2
FEAT16.3	Set Point Button	The user will have the ability to increase or decrease the set point by pressing a "change set point" button.	2.292	0	2
FEAT16.4	Change Set Point Dialog	Pressing the "change set point" button will display a change set point dialog with five buttons: "up", "down", and "OK" "Cancel" and "Default".	2.292	0	2

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT16.5	Change Set Point	Once the "change set point" dialog is active, the proposed set point will be displayed within the dialog box and can be increased or decreased by pushing an "up" or a "down" button.	0.655	3	2
FEAT16.5.1	Set Point Increment	Each press will change the set point by 5 mmHg.	0.655	1	2
FEAT16.5.2	Set Point Limits	The set point will have limits of 60 - 120 mmHg.	0.655	1	2
FEAT16.5.3	Set Point OK button	Pressing the OK button will activate the set point and close the dialog and display the set point button	0.982	9	2
FEAT16.5.4	Log Set Point Change	When a set-point change is activated the change will be recorded to the log file.	0.655	1	2
FEAT16.5.5	Set Point Cancel Button	Pressing the Cancel button will close the dialog box and leave the set point unchanged.	0.327	1	2
FEAT16.5.6	Set Point Default Button	Pressing the Default button returns the proposed set point to the default in the dialog box.	0.982	1	2
FEAT16.6	Set Point Changes Anytime	The set point can be changed at any time during resuscitation.	2.292	0	2

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT16.7	Set Point Dialog Visabilty	The change set point button and change set point dialog box will not be available when any other dialog box is open (TAC or Override). If the set point button or dialog is removed because another higher priority dialog box was displayed, it will be displayed and made available again when the higher priority dialog boxes have been closed. However, if auto control is terminated while the set point dialog is "hidden", it will not be redisplayed.	1.31	18	2
FEAT17	AC Button Selected	If the 'Start Auto-control' button is selected,	1.403	0	4
FEAT17.1	AC Initial Flow Rate	The CARA will initialize the pump at a default flow rate of 4 l/hr.	3.274	0	4
FEAT17.2	AC Inflate Cuff	Upon entering auto-control mode CARA will inflate the blood pressure cuff.	1.637	9	3
FEAT17.3	AC Cuff Not Available	If cuff pressures are not available	0.702	0	3
FEAT17.3.1	AC Cuff Not Avail Msg	An appropriate message should be displayed.	0.935	0	2
FEAT17.3.2	AC Cuff Not Avail Level 1 Alarm	A level 1 alarm is issued	0.935	3	4
FEAT17.3.3	AC Cuff Not Avail Override Buttons	An override "yes" button and an override "no" button will then be displayed.	0.935	4	2
FEAT17.3.3.1	AC Cuff Not Avail Override Yes Button	Pressing the override "yes" button will force the CARA to use the top priority (req. 13) uncorroborated pressure source for control.	0.702	4	3

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT17.3.3.1.1	AC Cuff Not Avail Override No Button	Pressing the "no" button will return to manual mode.	0.702	7	3
FEAT17.3.3.2	AC Cuff Not Avail Override Logged	A notation should be entered in the resuscitation file as to the button pushed.	1.403	1	3
FEAT17.3.4	AC Cuff Not Avail Alarm Reset	Pressing the alarm reset button will remove the audible and visual alarm and reattempt to inflate the cuff	0.935	9	4
FEAT17.4	AC Initiating Logged	A notation should be made in the resuscitation file, 'initiating auto-control'	1.637	9	3
FEAT17.5	AC Initiating Displayed	A notation should be made to the display, 'initiating auto-control'	1.403	12	2
FEAT17.6	No Available BPs Revert to Manual Mode	If cuff pressure is not available and there are no other blood pressures sources available CARA should revert to manual mode	3.508	3	5
FEAT17.6.1	No Available BPs Revert to Manual Mode Display	An appropriate message should be displayed.	1.637	0	2
FEAT17.6.2	No Available BPs Revert to Manual Mode Alarm and Log	A level 1 alarm is issued (logged to file and displayed)	1.637	3	4
FEAT18	Valid BP Range	Mean pressure readings from the Propaq must be within 40 - 150 mmHg to be valid throughout resuscitation.	7.016	33	2
FEAT19	No Propaq Data Stream	If the CARA does not receive the data stream from the Propaq	1.403	9	2
FEAT19.1	No Propaq Data Stream Display	An appropriate message should be displayed.	1.403	0	2

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT19.2	No Propaq Data Stream Level 2 Alarm	A level 2 alarm should be issued	1.871	3	3
FEAT20	BP Corroboration Begins	Once the 'Start Auto-control' button has been pressed, BP source corroboration begins in the order of source priority (req. 13), if a cuff pressure is available	0.935	3	3
FEAT20.1	BP Corroboration Source Control	A source control pressure will be compared to a corresponding cuff pressure	1.684	0	3
FEAT20.2	BP Corroboration Source Control Within 10%	If the source pressure is within 10% of the corresponding cuff pressure the source is corroborated and will be used for control	1.684	0	3
FEAT20.3	BP Corroboration Source Control Not Within 10%	If the source pressure is not within 10% of the corresponding cuff pressure, two more cuff readings will be taken and compared to corresponding source readings	0.561	0	3
FEAT20.3.1	BP Corroborated	If both source pressure readings are within 10% of the corresponding cuff readings, the source is corroborated and will be used for control	0.561	0	3
FEAT20.3.2	BP Corroboration Failure Dialog	If both source pressure readings are not within 10% of the corresponding cuff readings, an override dialog box will be displayed and the corroboration failure will be logged.	0.561	9	2
FEAT20.3.2.1	BP Corroboration Failure Dialog Yes Button	If the override "YES" button is pressed, the uncorroborated source will be used for control	0.561	0	2

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT20.3.2.2	BP Corroboration Failure Dialog No Button	If the override "NO" button is pressed, CARA will attempt to corroborate the next priority source (req. 13) based on readings already collected	0.561	0	2
FEAT20.3.2.2.1	BP Corroboration Failure Cuff for Control	If no other source is available, CARA will use the cuff pressure for control.	0.748	0	2
FEAT20.3.2.2.2	BP Corroboration Failure Additional Attempts	Uncorroborated blood pressures sources will be compared to each cuff reading. If the readings are within 10% the CARA will automatically switch over to the new source. Override dialog boxes will not be displayed for these subsequent corroboration attempts.	0.748	9	2
FEAT20.4	BP Corroboration Cuff for Control	During source corroboration, CARA will use the cuff pressure for control	1.123	3	3
FEAT20.5	Display Control Source	CARA should display the blood pressure control source (e.g. Arterial Line, Pulse Wave, Cuff, etc.)	1.123	3	2
FEAT20.6	Log Control Source	CARA will log the blood pressure control source to the log file	0.935	0	3
FEAT20.7	Re-corroborate Interval	CARA will re-corroborate the blood pressure control source with the cuff every 30 minutes.	0.935	24	2
FEAT20.7.1	Re-corroborate Wait Condition	Any active corroboration attempt must be completed before the periodic 30-minute re-corroboration can begin.	0.935	24	2
FEAT20.7.2	No Cuff for Re-calibration	If the cuff pressure is not available for re-calibration,	0.748	4	2
FEAT20.7.2.1	No Cuff for Re-calibration Display	An appropriate message should be displayed	0.748	2	2

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT20.7.2.2	No Cuff for Re-calibration Level 1 Alarm	A level-1 alarm should be issued	0.748	2	4
FEAT20.8	Higher BP Source Available	If a higher priority blood pressure source than the one that CARA is using becomes available, CARA should corroborate the higher priority blood pressure source using the current blood pressure source.	1.684	12	3
FEAT20.8.1	Higher BP Source Available Deconfliction	If a source is in the process of being corroborated, or an override question is pending and a new higher priority source begins reporting, corroboration of the new higher priority source cannot begin until the current corroboration process complete or the override question is answered.	1.123	18	2
FEAT21	Deleted Rqt	Deleted Rqt	0	0	0
FEAT22	Deleted Rqt	Deleted Rqt	0	9	0
FEAT23	A-Line Not Available	If an arterial line is not available then other blood pressure sources should be used.	7.016	9	3
FEAT24	Pulse Wave Available	If the pulse wave signal is detected,	0.935	2	3
FEAT24.1	Pulse Wave Calibration	CARA should immediately begin to calibrate the pulse wave using an average of 3 cuff pressures taken one minute apart. It is expected that a valid pulse wave pressure reading will be available every 15 seconds for control purposes. To be used, the pulse wave must be calibrated using the cuff pressure.	1.754	5	3

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT24.2	Pulse Wave Re-calibration Interval	The pulse wave should be re-calibrated every 15 minutes using the average of two cuff pressures taken one minute apart.	1.403	2	3
FEAT24.3	Pulse Wave Re-calibration w/ No Cuff	If the cuff pressure is not available for re-calibration,	1.403	2	3
FEAT24.3.1	Pulse Wave Re-calibration w/ No Cuff Display	An appropriate message should be displayed.	0.935	2	2
FEAT24.3.2	Pulse Wave Re-calibration w/ No Cuff Level 1 Alarm	A level 1 alarm is issued	1.754	5	4
FEAT24.3.3	Use Last Good Pulse Wave	The CARA should continue using the pulse wave with the last good calibration.	1.754	2	3
FEAT25	Only Cuff to be Used	If only a cuff pressure is to be used the CARA should immediately initiate 5 blood pressure readings one minute apart.	1.754	0	3
FEAT26	CARA Re-adjust after Each New BP	The CARA will readjust after each blood pressure reading.	7.016	0	3
FEAT27	Cuff As Control Interval	When the cuff pressure is being used for control, CARA should set a cuff reading frequency based on a table. In general, blood pressures will be taken more frequently while below the set point. If the cuff is already inflating for some other reason when the time arrives for another reading, an additional cuff reading does not need to be requested.	1.871	15	3

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT27.1	Cuff As Control Interval, BP 60 or Below	If the mean BP is 60 or below, cuff pressures will be taken once per minute.	1.871	10	3
FEAT27.2	Cuff As Control Interval, BP (60-70]	If the mean BP is (60 - 70], cuff pressures will be taken once every 2 minutes.	1.871	10	3
FEAT27.3	Cuff As Control Interval, BP (70-90]	If the mean BP is (70 - 90], cuff pressures will be taken once every 5 minutes.	1.871	10	3
FEAT27.4	Cuff As Control Interval, BP Above 90	If the mean BP is above 90, cuff pressures will be taken once every 10 minutes.	1.871	10	3
FEAT28	No Valid BP in 3 Minutes, Revert to Manual Mode	If CARA can not obtain a valid blood pressure in 3 minutes, it should revert back to manual mode.	3.742	6	4
FEAT28.1	No Valid BP in 3 Minutes, Revert to Manual Mode Display	An appropriate message should be displayed	2.806	1	2
FEAT28.2	No Valid BP in 3 Minutes, Revert to Manual Mode Level 2 Alarm	A level 2 alarm should be issued.	2.806	3	3
FEAT29	Calculate Pump Voltage	Once a valid blood pressure has been established the CARA should calculate a voltage to drive the pump.	7.016	0	5
FEAT29.1	Display AC Mode	A notation indicating that the system is in auto-control mode should be made on the display	1.403	0	2
FEAT29.2	Log AC Mode	A notation indicating that the system is in auto-control mode should be made in the resuscitation file.	2.806	0	3

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT29.3	Display BP Set Point for AC Mode	Also, a horizontal line indicating the blood pressure set point should be shown on the graphical display.	2.806	0	2
FEAT30	New Voltage Interval	When using an arterial line, pulse wave or other beat-to-beat blood pressure a new voltage should be calculated every 15 seconds by the CARA.	7.016	0	4
FEAT30.1	New Voltage Interval for Cuff	When using the cuff pressure a new control voltage should be established after every blood pressure reading.	7.016	0	3
FEAT31	Minimum Flow Rate	The CARA will always maintain at least a KVO flow rate.	14.032	9	5
FEAT32	BP Source Change Monitoring	CARA will always monitor for blood pressure source changes	2.339	0	3
FEAT32.1	Corroborate Higher Priority Source	If the new source is a higher priority source than the current control source (req. 13) the new source will be corroborated (req. 20)	2.923	0	3
FEAT32.1.1	Use New Corroborated Source	If the new source is corroborated, CARA should change to use the new source for control	4.093	0	3
FEAT32.1.2	Not Corroborated Override Dialog	If the new source is not corroborated, an override dialog should be displayed to give an opportunity to change to the uncorroborated higher priority source, as described in req. 20. An override alert is issued at this time also (an alarm).	2.339	0	3
FEAT33	Deleted Rqt	Deleted Rqt	0	9	0
FEAT34	Terminate AC Button Availability	When the CARA is in auto-control mode a 'Terminate Autocontrol' button should be made available.	16.37	9	5

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT35	Deleted Rqt	Deleted Rqt	0	9	0
FEAT36	Deleted Rqt	Deleted Rqt	0	9	0
FEAT37	Deleted Rqt	Deleted Rqt	0	9	0
FEAT38	Deleted Rqt	Deleted Rqt	0	15	0
FEAT39	Falling BP Monitoring	CARA should monitor for falling blood pressure	4.677	33	4
FEAT39.1	Falling BP Display	An appropriate message should be displayed	3.508	10	2
FEAT39.2	Falling BP Level 2 Alarm	A level 2 alarm should be issued.	3.508	13	3
FEAT40	Deleted Rqt	Deleted Rqt	0	12	0
FEAT41	Deleted Rqt	Deleted Rqt	0	18	0
FEAT42	Lost BP in AC Mode	While in auto-control mode, if a beat-to-beat blood pressure signal is lost for more than 1 minute	2.339	0	4
FEAT42.1	Lost BP in AC Mode Display	An appropriate message should be displayed	2.339	0	2
FEAT42.2	Lost BP in AC Mode Level 1 Alarm	A level-1 alarm should sound	2.806	0	4
FEAT42.3	Lost BP in AC Mode Graphical Display Notation	Notation should be made on the graphical display.	1.871	0	2
FEAT43	Beat-to-Beat Signal Loss	If the beat-to-beat signal is lost for more than 3 minutes	1.871	3	4
FEAT43.1	Beat-to-Beat Signal Loss Display	An appropriate message should be displayed	0.935	1	2
FEAT43.2	Beat-to-Beat Signal Loss Level 2 Alarm	A level 2 alarm should sound	1.871	4	3
FEAT43.3	Beat-to-Beat Signal Loss Drive Voltage Setting	The drive voltage should be set to the last good blood pressure	2.806	1	5

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT43.4	Beat-to-Beat Signal Loss Revert to Secondary BP Source	CARA should then revert to using secondary blood pressure sources using appropriate quality control procedures.	2.806	3	3
FEAT44	Cuff BP Invalid Reading	If only the cuff pressure is being used and an expected blood pressure reading is invalid	1.169	9	3
FEAT44.1	Cuff BP Invalid Reading Display	An appropriate message should be displayed	1.169	3	2
FEAT44.2	Cuff BP Invalid Reading Level 1 Alarm	A level-1 alarm should sound	2.105	3	4
FEAT44.3	Initiate New Cuff BP Reading	CARA should then initiate another request for a cuff pressure.	1.169	3	3
FEAT44.3.1	If New Cuff BP Invalid	If this pressure is invalid,	0.585	3	3
FEAT44.3.1.1	If New Cuff BP Invalid Display	An appropriate message should be displayed	1.169	3	2
FEAT44.3.1.2	If New Cuff BP Invalid Level 2 Alarm	A level-2 alarm should sound	1.403	3	3
FEAT44.3.1.3	If New Cuff BP Invalid Revert to Manual Mode	The system will revert to manual mode	1.52	12	5
FEAT44.4	Log Cuff BP Invalid	Notations should be made to the resuscitation file	1.403	3	3
FEAT45	Alarm Description	Alarms consist of an audible alarm and a visual alarm message. When an alarm becomes active, the audible alarm sounds and the alarm message is placed on the display.	0.655	3	4

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT45.1	Alarm Msg Priority	Alarm messages will be listed according to alarm priority as described in the alarm table.	1.146	10	4
FEAT45.1.1	Alarm Message Description	The alarm messages should be in the form of directions to the caregiver on how to fix the problem. If multiple options are available to fix the problem, then all fixes should be listed in priority in a single message on the display.	1.637	10	3
FEAT45.2	Active Alarms When Pump Un-plugged	If the pump is unplugged while any alarms are active, the active alarms should be automatically reset. Only the pump-unplugged alarm will remain or become active if appropriate.	1.473	4	3
FEAT45.3	Alarm Reset When Condition Resolved	Alarms will automatically be reset and the alarm message will be removed if CARA detects that the alarm condition has been resolved.	1.473	4	3
FEAT45.4	Alarm Buttons	Two "soft" buttons will appear whenever there is an alarm. One button will allow temporary silencing of the alarm for a set period. The other button will be an "Acknowledge/reset " button.	0.819	5	3
FEAT45.4.1	Silence Alarms button	Caregiver should have the ability to silence the audio alarm while fixing the problem by pressing the "silence alarms" button.	0.819	7	3
FEAT45.4.1.1	Silence Alarms Period	The alarm should be silenced only for a period of time. Time period depends on the nature of the problem as described in the alarm table.	1.146	7	4

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT45.4.1.2	Silence Alarms Display	The alarm message should remain on the display when the temporary silence button is pushed.	1.146	13	4
FEAT45.4.1.3	Silence Alarms Not Resolved	If the alarm condition is not fixed and time expires, then the audible alarm should sound again.	1.146	13	4
FEAT45.4.2	Alarm Acknowledge /Reset Button	Once the caregiver has executed the desired fix, he will push the Acknowledge/Reset button (this is necessary only for conditions that must be polled by the CARA system)	0.655	7	3
FEAT45.4.2.1	Fault Fixed TS Timer Disabled	If the fault is fixed, then the alarm will be completely reset and the temporary silence "TS" timer will be disabled.	1.31	5	4
FEAT45.4.2.2	Fault Not Fixed, Alarm Repeat	If the fault is not fixed, the audible alarm will immediately sound and the visual portion of the alarm will continue and the message will stay on the display. The temporary alarm silence timer will be removed.	1.31	31	4
FEAT45.5	No Permanent Alarm Off Function	There shall be no software provision for turning the alarms off permanently.	1.637	4	4
FEAT46	All Alarms Logged	All alarms will be recorded in the resuscitation file	3.742	0	3
FEAT46.1	All Alarms Logged with time	The time issued will be recorded	2.806	0	3
FEAT46.2	All Alarm Resets Logged with Time	The time reset will be recorded	2.806	0	3
FEAT47	Pump Unplugged Logged with Timestamp	If the pump is unplugged an entry should be made to the log file with a timestamp.	1.403	12	3

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT47.1	When No BackEMF	While there is no back EMF detected	0.819	4	3
FEAT47.1.1	When No BackEMF Display	The system should display an appropriate message.	1.403	4	2
FEAT47.2	When No BackEMF Detected	While back EMF is detected	0.819	4	2
FEAT47.2.1	When No BackEMF Detected Display	A message on the display should appear	1.988	4	2
FEAT47.2.2	When No BackEMF Detected Level 1 Alarm	A level 1 alarm should be issued	2.339	8	4
FEAT47.3	Pump Unplugged during AC	During auto-control mode,	0.585	12	3
FEAT47.3.1	Deleted Rqt	Deleted Requirement	0	12	0
FEAT47.3.2	Deleted Rqt	Deleted Requirement	0	12	0
FEAT47.3.3	Pump Unplugged during AC, Terminate AC	CARA should exit auto-control mode	2.339	13	5
FEAT48	Terminate AC Selected	Whenever the 'Terminate auto-control' button is selected the auto-control termination sequence begins.	1.286	3	5
FEAT48.1	Deleted Rqt	Deleted Requirement	0	9	0
FEAT48.2	Terminate AC Confirmation Message	A confirmation message indicating that control will be released should be displayed in a dialog box. This dialog box will take priority over all other open dialogs. Any other open dialog boxes will be closed when the Terminate Auto-control dialog box is displayed.	3.859	3	2

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT48.3	Terminate AC Yes Confirmation Button	A confirmation yes button will be displayed	2.572	0	3
FEAT48.3.1	Terminate AC Yes Confirmation Button Selected	Pressing this button will relinquish flow control of the pump. The pump will operate at its hardware switch setting.	5.145	0	5
FEAT48.3.1.1	Terminate AC Yes Confirmation Button Selected Display	A notation that the system is in manual mode should be made to the display	2.572	0	2
FEAT48.3.1.2	Terminate AC Yes Confirmation Button Selected Logged	A notation that the system is in manual mode should be made to the resuscitation file.	5.145	0	3
FEAT48.4	Terminate AC No Confirmation Button	A confirmation no button will be displayed.	1.286	0	2
FEAT48.4.1	Terminate AC No Confirmation Button Selected	Pressing the this button will return the system to Auto-control, closing the Terminate auto-control dialog box, and re-opening the highest priority pending dialog box, if any exist.	1.286	0	3
FEAT48.5	Terminate AC Confirmation Dialog, Change Set Point Button	While the confirmation dialog box is displayed, the Change Set Point button should be disabled if it is available, and the Alarm rest button should be disabled if it is available.	2.572	18	3
FEAT49	Action Button Action	When an action button is pressed the button should be made unavailable (removed or disabled).	7.016	9	2

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT50	Interrupt Signals	Logic level input signals (Pump connection, continuity, Occlusion) will occur as an interrupt signal when the state of a signal changes.	7.016	0	2
FEAT51	Clock Interrupts	A clock interrupt must trigger certain events at even five-second and 60-second clock intervals after the pump is plugged in.	2.105	6	2
FEAT51.1	5 Second Clock Interrupts	At the five-second interval the sequence of events should be to check EMF, display the updated flow rate, and then check impedance value.	2.806	9	2
FEAT51.2	60 Second Clock Interrupts	At the 60-second interval the sequence of events should be to check EMF, display the updated flow rate, check the impedance value, write the flow rate, cumulative volume infused and impedance value to the log file.	2.806	9	2
FEAT52	Impedance Polling Sequence	The impedance will be polled immediately after the EMF when the pump is plugged in and then on every even 5-second interval while the pump remains plugged in.	2.385	34	2
FEAT52.1	Impedance Value Logged	The impedance value should be logged once per minute on the minute.	2.315	24	2
FEAT52.2	AirOK Fault, Read Impedance and Log	If the Air OK fault occurs, the impedance should be read immediately (at the time of the fault) and logged to the file.	2.315	39	3

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT53	Alarm Priorities and Silencing Times	Alarm priorities and silencing times table (Increment 1) Pri Alarm Silence time 1 Pump unplugged 2 during manual mode 2 2 Continuity fault 5 3 Air lock detected 2 4 Occlusion 2 TBD Polling failure inf TBD Data log failure N/A	11.693	0	4
FEAT54	Polling Request Failure Actions	If a polling request fails, CARA should retry the request at one second intervals until successful for a maximum of three readings.	2.105	18	3
FEAT54.1	Polling Request Failure Retries and Alarm	If the maximum number of retry attempts is made with no success, CARA should issue an alarm and message stating that a data acquisition failure has occurred.	1.754	9	4
FEAT54.1.1	No Impedance in AC, Terminate AC	If impedance reading cannot be obtained while in auto- control mode, CARA should exit auto-control mode	3.157	15	5
FEAT55	Log File Failure Alert Display	If an attempt to write to the log file fails, CARA should display an alert message indicating that a data logging failure has occurred and continue operating.	2.339	9	3
FEAT56	Pump Status Changes Timestamp	When the pump status changes, an entry with a timestamp should be made to the log file to record the change in status. These log messages should correspond to the display message in req. 2. (This requirement is redundant in some cases)	2.339	0	3

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT57	Pump Startup condition	A1 - Assume the pump is not in at startup (see Q62) and that the system will issue an interrupt when the status changes.	4.677	0	2
FEAT58	Alarm Reset Availability	A3 - Alarm reset buttons are available only when an alarm has been triggered (alarm buttons are soft buttons) - (see Q64)	2.339	0	4
FEAT59	AirOK and Continuity Startup States	A6 - Assume the occlusion, Air-OK, and continuity status are OK when the pump is plugged in (see Q62) and that the system will issue an interrupt when the status changes.	4.677	0	2
FEAT60	Polling Request Interruption	A15 - Polling requests and event service should not be interrupted. The event service will complete before other inputs are handled. (System interrupts will still occur, but input from them will be queued into the CARA)	2.339	9	3
FEAT61	Flow Rate Calculation Sequence	A16 - flow rate is calculated and displayed immediately when the pump is plugged in and the back EMF value has been read	7.016	0	3
FEAT62	Data Logging Failure Additional Attempts	A17 - If a data logging failure occurs, data logging attempts should continue as normal	2.339	0	3
FEAT63	Pump Unplugged During Active Alarms	A18 - If the pump unplugged during operation alarm is active and the alarm reset button is pressed, the alarm will be cleared and rest if the pump is still not present (system will return to initial state with no pump).	2.339	0	4

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT64	Polling Failure and Future Polling Attempts	A19 - If a polling failure occurs, polling attempts will continue as normal	2.339	0	3
FEAT65	Alarm Reset and Polling Device Interaction	A20 - If an alarm reset button is pressed and a polling device was the source of an alarm, CARA will immediately poll the appropriate devices in order by device or alarm priority.	2.339	0	4
FEAT66	Logging Sequence	A21 - Logging occurs as the last action in an event service generally. Alarms, however, are logged immediately when the alarm is issued if data logging is available.	2.339	0	3
FEAT67	Alarm Silence Disable Button	A22 - If the alarm silence button is pressed, the alarm silence button will be disabled until a new audible alarm is active	2.339	0	4
FEAT68	Multiple Alarm Conditions During an Alarm Silence	A23 - If the current alarms are silenced and a new alarm condition occurs an audible alarm will be issued for the new alarm condition.	2.339	0	4
FEAT69	Polling Failure and Data Logging	A24 - If a polling failure occurs data logging will continue and note that the polled value is unknown	2.339	0	3
FEAT70	Dialog Box Priority	Only one dialog box can be displayed at a time. If multiple dialog boxes are pending, the highest priority dialog will be displayed.	1.169	9	2
FEAT70.1	Dialog Box Priority Order	The dialog box priority is (from high to low) Terminate Auto-control Source Override Change Set Point	1.169	9	2

FEAT Tag	Name	Requirement Text	AHP Priority	Rqts Clarity	Safety
FEAT70.2	Dialog Box Priority Display Changes	If a dialog box is currently displayed and the system conditions change to require a higher priority dialog box to be displayed, the original dialog box will be closed and the higher priority dialog will be displayed.	1.169	9	2
FEAT70.2.1	Dialog Box Re-Display	When the higher priority dialog box is closed, the lower priority dialog box will be re-displayed, unless Auto-control is terminated.	1.169	9	2

APPENDIX C: SEATOOLS CARA MODEL DESCRIPTIONS

A. INTRODUCTION

This appendix provides an overview of the SEATools model of the CARA Infusion Pump software [WRAI01a, b, c]. This model was designed based on the requirement set of the CARA listed in Appendix B. The interoperability between this model and requirement set provide the confirming evidence for the dissertation hypothesis.

B. VERSION 1

Version 1 (actually v0.1 in the SEATools directory) of the CARA model is a non-working, abstract view of the overall architecture implemented in version 2 (v0.2). Its only purpose is to give someone unfamiliar with the more detailed model in version 2 a high level overview of the major constructs in the model architecture.

1. Parent Vertex: Puett_Liang_CARA

The overall system environment consists of four main components: The *Patient*, the *LSTAT* stretcher, the *Infusion_Pump*, and the *CARA* Software System (see Figure 100 below). Later in version 2, the patient is removed from the model and the infusion pump and LSTAT are modeled as external simulations. The major system flows in this vertex includes the following: *fluid* from the pump to the patient, blood pressures (*bps*) from the patient to the LSTAT (which then passes them on to the CARA), inputs from CARA to the LSTAT and pump (*LSTAT_commands* and *pump_commands*), and finally sensor readings from the LSTAT and pump to CARA (*LSTAT_status* and *pump_sensor_readings*).

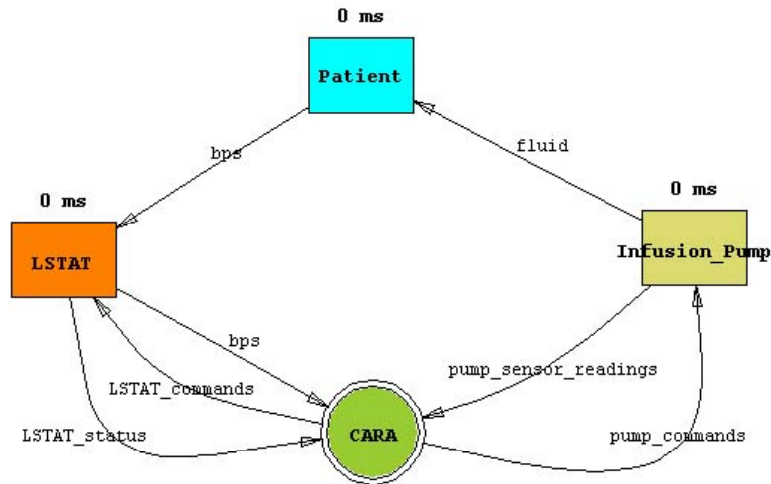


Figure 100 Top Level CARA Model v1

2. Parent Vertex: CARA

This level of version 1 shows the three main modules of the design: a *Pump_Control_Module* with a main function of resolving the blood pressure to use and determining the appropriate flow rate for the pump, an *IO_Module* with a main function of sending and receiving inputs to the CARA operator via the display, and the *Management_Module* with main functions of monitoring the status of the pump, the lines, and the system and for logging data into the resuscitation file (see Figure 101 below).

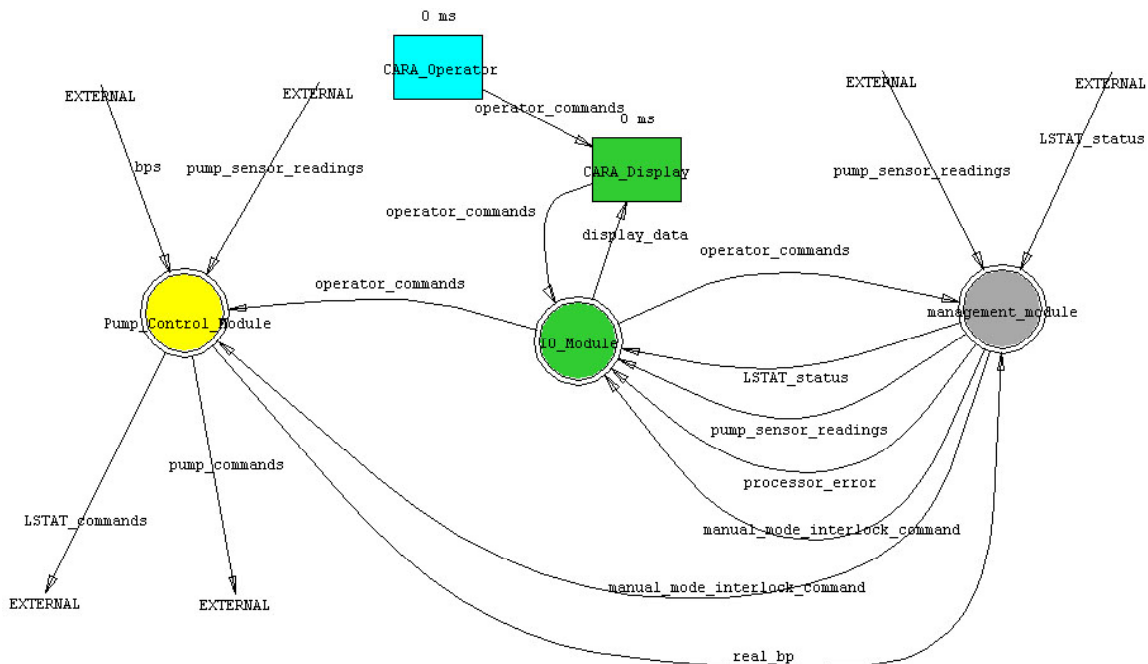


Figure 101 CARA Software Model v1

3. Parent Vertex: Management_Module

Central to this module is a *manual_mode_interlock* that is the primary operator responsible for returning the system to a manual mode in case of failure of any component. Feeding this operator is data from the *line_monitor* that monitors the sensor readings from the pump and LSTAT. Also feeding the *manual_mode_interlock* is a *processor_watchdog*. This watchdog would be implemented on a separate processor from the main processor handling the bulk of the operations and is responsible for sensing any major processor failure and then alerting the operator (via the display). One final element of the *Management_Module* is the *Resuscitation_File* where all information about the changing state of the CARA system is recorded (see Figure 102 below).

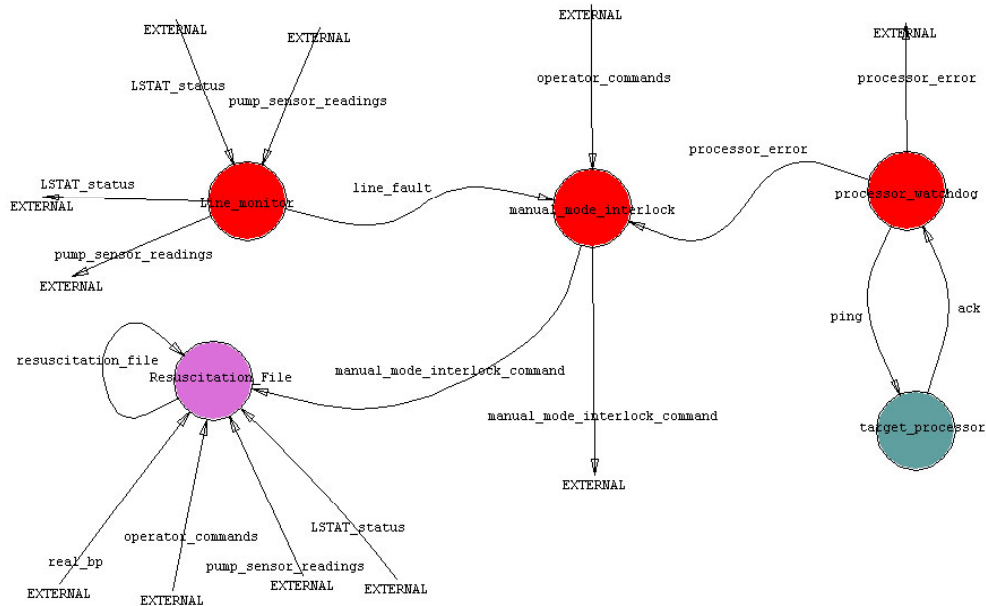


Figure 102 Management Module v1

4. Parent Vertex: Pump_Control_Module

This is the major safety critical module of the CARA. It is responsible for resolving what blood pressure to use and for determining the correct input to the pump when the system is in auto-control mode (see Figure 103 below). Because of the safety critical nature of this module, we chose to implement this with Triple Modular Redundancy (TMR). This specific safety architecture was not called for explicitly in the requirement statement; however, the safety environment implicitly requires some form of redundancy to ensure that the proper commands are sent to the pump. The TMR architecture uses three concurrent modules performing similar functions and producing similar output, but using different internal algorithms in their calculations. A voting element is then responsible for determining which output to use.

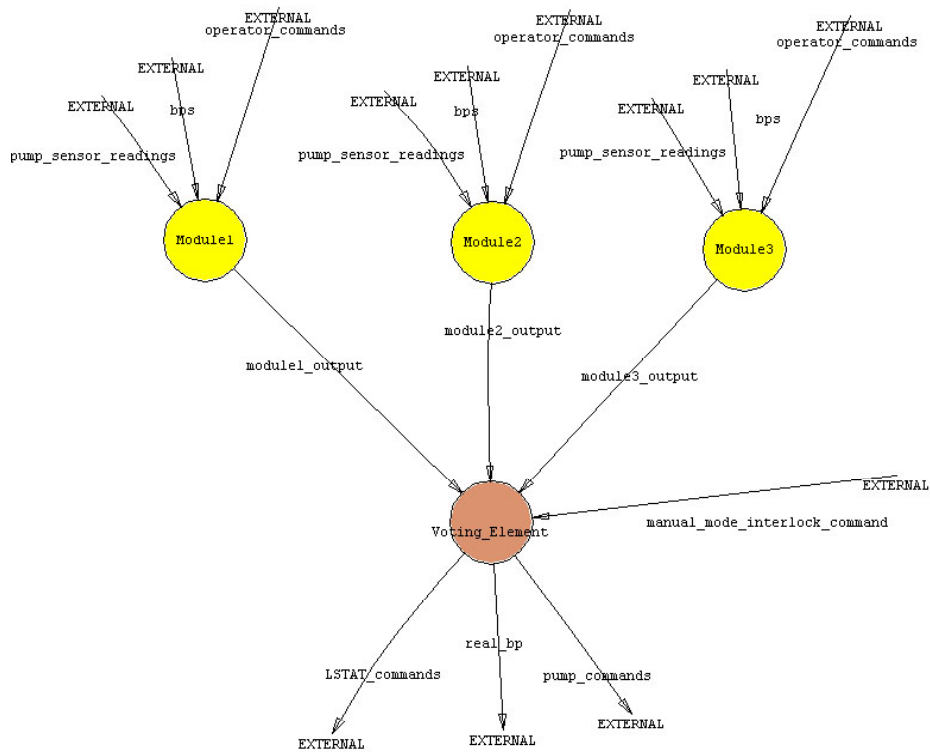


Figure 103 Pump_Control Module v1

5. Parent Vertex: IO_Module

This module handles the input and output to the CARA display for the benefit of the CARA operator (see Figure 104 below). The alarm functions have been separated from other display functions to help isolate the safety critical functions. Also note the duplicate *alarm_controller1* and *2*. This additional *alarm_controller* would be implemented on the second processor with the processor watchdog so that in case of processor failure, alarms would be raised to the display.

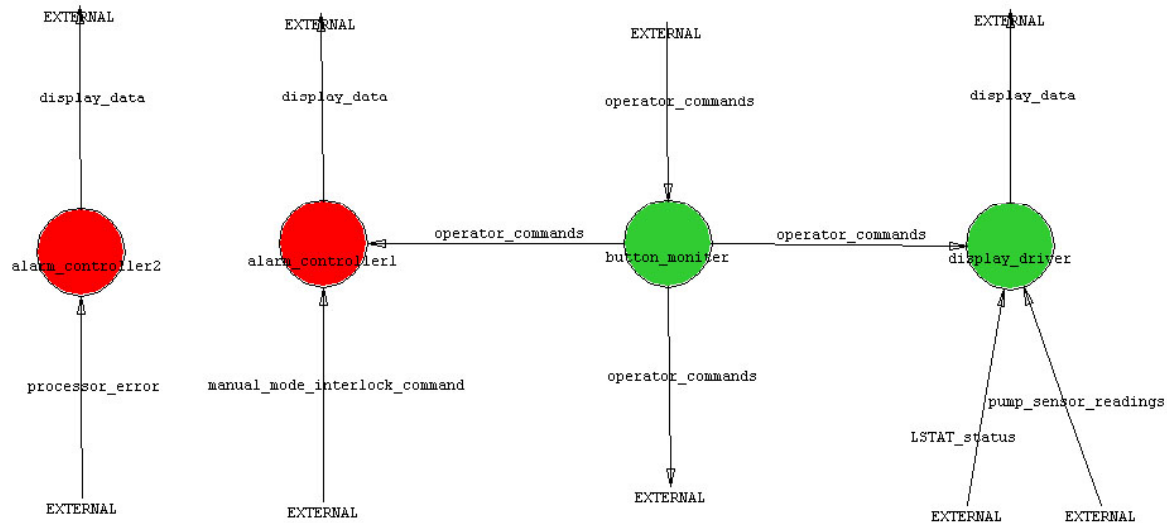


Figure 104 IO_Module v1

B. VERSION 2

Version 2 (actually v0.2 within SEATools) is the working prototype that implements a majority of the CARA requirements specified in Segment 3 of the CARA Requirement Statement [WRAI01c]. This version is much more detailed than version 1 and is described in many more layers of decomposition. The "human" element (the patient and CARA operator) has been removed from the model in this version. The external interfaces to the system are modeled as simulators for the LSTAT, the Infusion Pump, and the CARA Display.

1. Parent Vertex: Puett_Liang_CARA

The overall system environment consists of just three main components: the *LSTAT* stretcher is assumed to provide the majority of patient related information (e.g. blood pressures), the *Infusion_Pump* is the main item for control, and the *CARA* Software System is the system driving the infusion pump based on data received from the LSTAT (see Figure 105 below). The major system flows in this vertex between the CARA and LSTAT are the following: three different kinds of blood pressures (*aline_bp*, *pulse_wave_bp*, *cuff_bp*), additional sensor data from the LSTAT (a signal that the LSTAT is turned on (*LSTAT_power_on*) and the status of the pump (*pump_plugged_in_status*)), and an output from CARA to the LSTAT (commands to inflate the blood pressure cuff (*inflate_cuff*)). The major system flows between the CARA and the Infusion Pump are the following: the *pump_speed* to the pump (a voltage), from the pump several safety related data items (*impedance*, *continuity_disrupted*, *occlusion_detected*, and *air_disrupted*), and from the pump the *back_EMF* which is proportional to the actual rate at which the pump is infusing liquid.

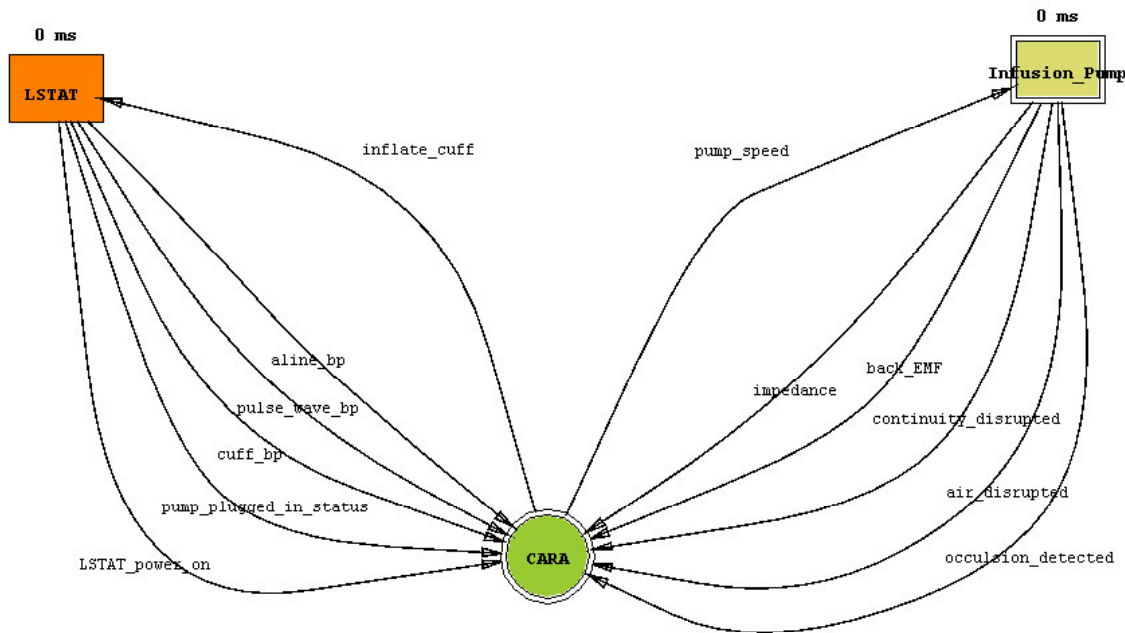


Figure 105 CARA Software Model v2

2. Parent Vertex: Infusion_Pump

This level shows the decomposition of the Infusion Pump. There is an eight pin ribbon cable between the pump and CARA. Eight of the terminators in this level correspond to the pins of the ribbon cable -- only 5 of which actually interact with CARA (*pin3_pump_speed_voltage*, *pin4_back_EMF*, *pin5_AirOK*, *pin7_OccOK*, and *pin8_impedance_signal*) (see Figure 106). The requirements also describe the need for continuity checking of all the pins via a hardware interlock -- this is modeled as an additional terminator (*continuity_interlock*).

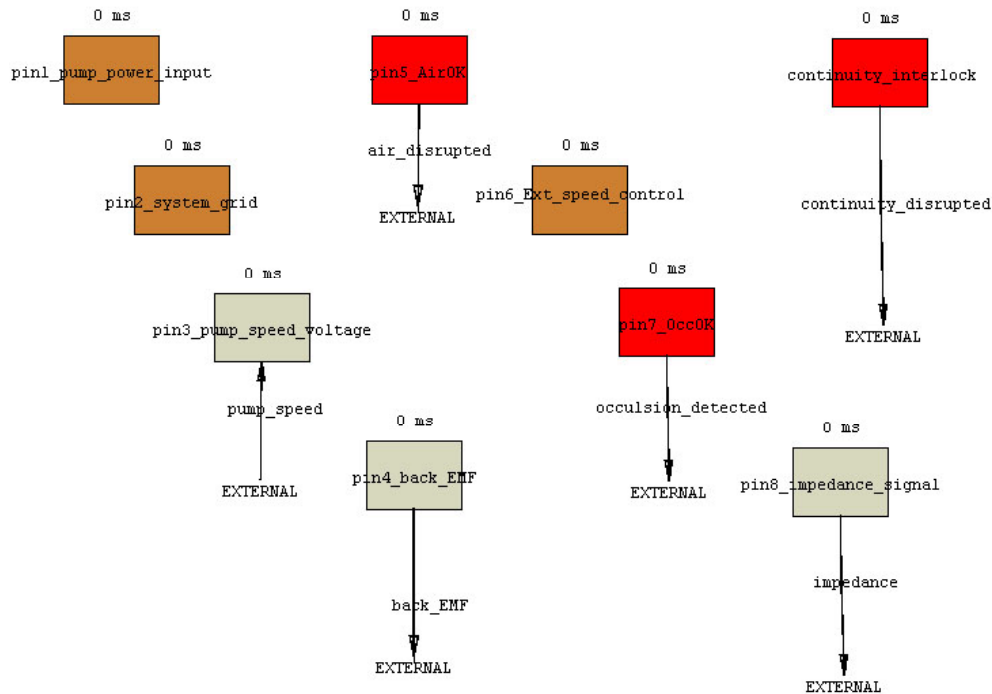


Figure 106 Infusion_Pump Pin-outs v2

3. Parent Vertex: CARA

This level shows the three main modules of the design: a *Pump_Control_Module* with a main function of resolving the blood pressure to use and determining the appropriate flow rate for the pump, an *IO_Module* with a main function of sending and receiving inputs to the CARA operator via the display, and the *Management_Module* with main functions of monitoring the status of the pump, the lines, and the system and for logging data into the historical resuscitation file (see Figure 107).

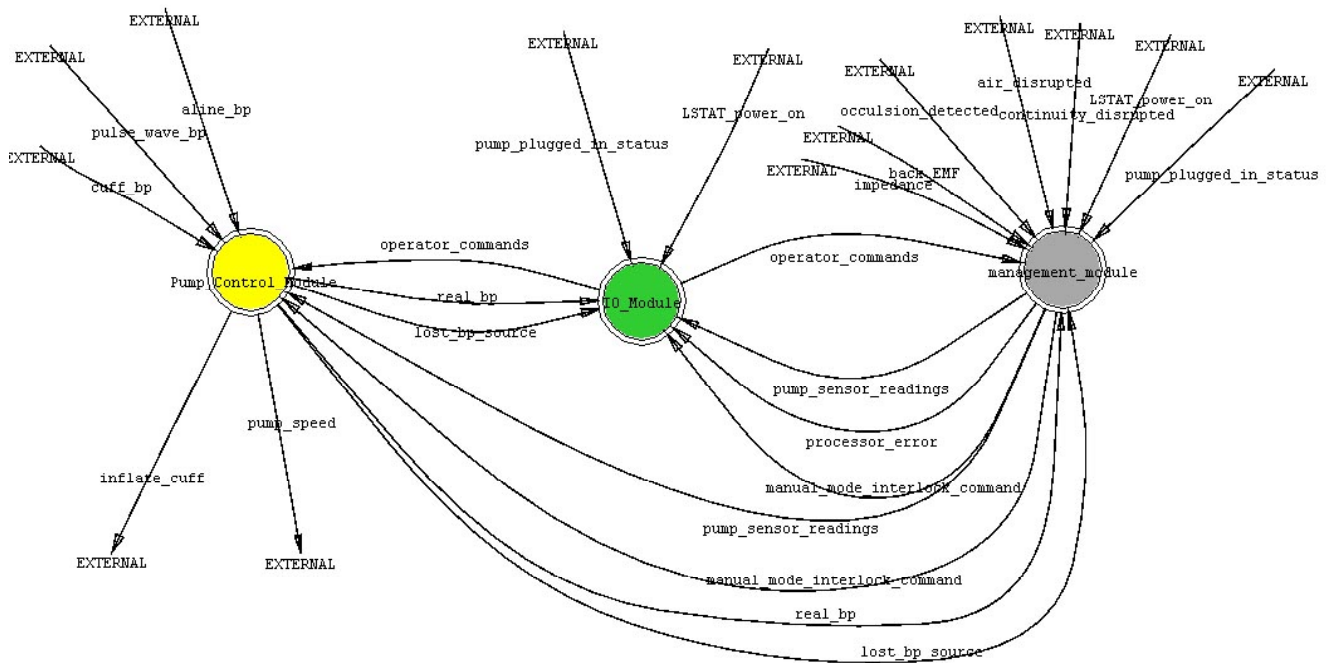


Figure 107 CARA System Modules v2

4. Parent Vertex: Management_Module

Central to this module is a *manual_mode_interlock* that is the primary operator responsible for returning the system to a manual mode in case of failure of any component (see Figure 108). Feeding this operator is data from the *line_monitor* that monitors the sensor readings from the pump and LSTAT. Also feeding the *manual_mode_interlock* is a processor watchdog. This watchdog would be implemented on a separate processor from the main processor handling the bulk of the operations and is responsible for sensing any major processor failure and then alerting the operator (via the display). One final element of the *Management_Module* is the *resuscitation_file* where all information about the changing state of the CARA system is recorded.

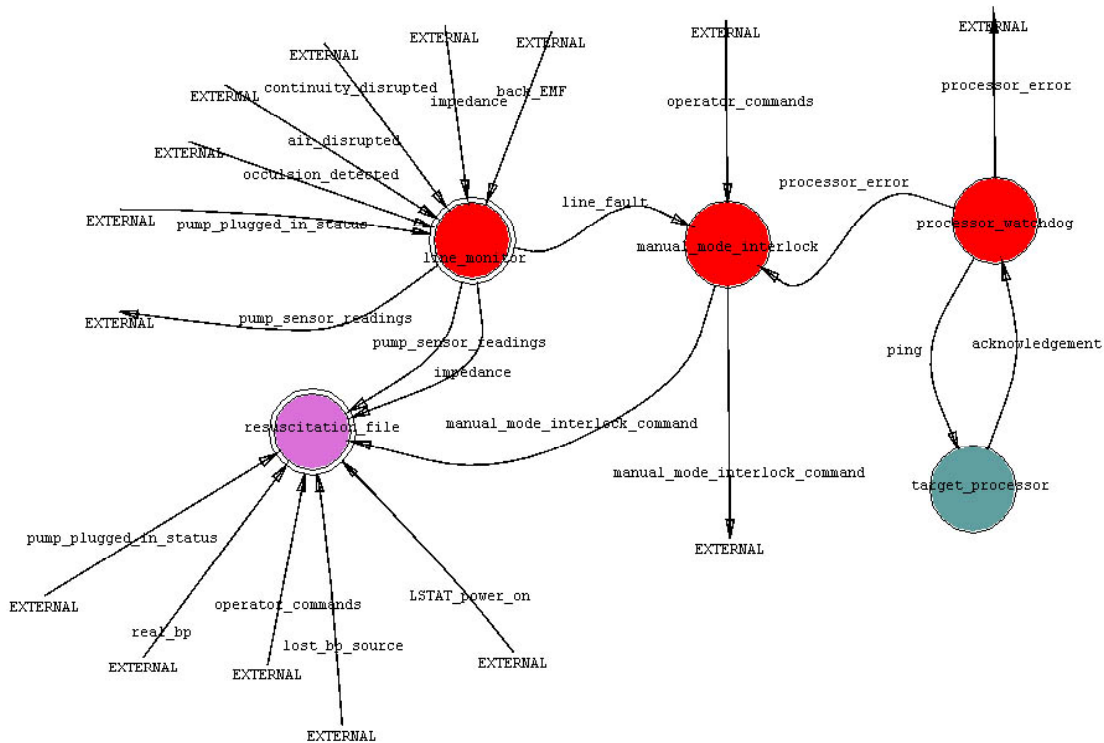


Figure 108 Management_Module v2

5. Parent Vertex: Pump_Control_Module

This is the major safety critical module of the CARA. It is responsible for resolving what blood pressure to use and for determining the correct input to the pump when the system is in auto-control mode. Because of the safety critical nature of this module, we chose to implement this with Triple Modular Redundancy (TMR) (see Figure 109). This specific safety architecture was not called for explicitly in the requirement statement; however, the safety environment implicitly requires some form of redundancy to ensure that the proper commands are sent to the pump. The TMR architecture uses three concurrent modules performing similar functions and producing similar output, but using different internal algorithms in their calculations. A voting element is then responsible for determining which output to use. *Module1* is the only module of the three that has been fully decomposed. *Modules2* & 3 could be decomposed similarly to *Module1* but would use different algorithms (inserted at the programming stage).

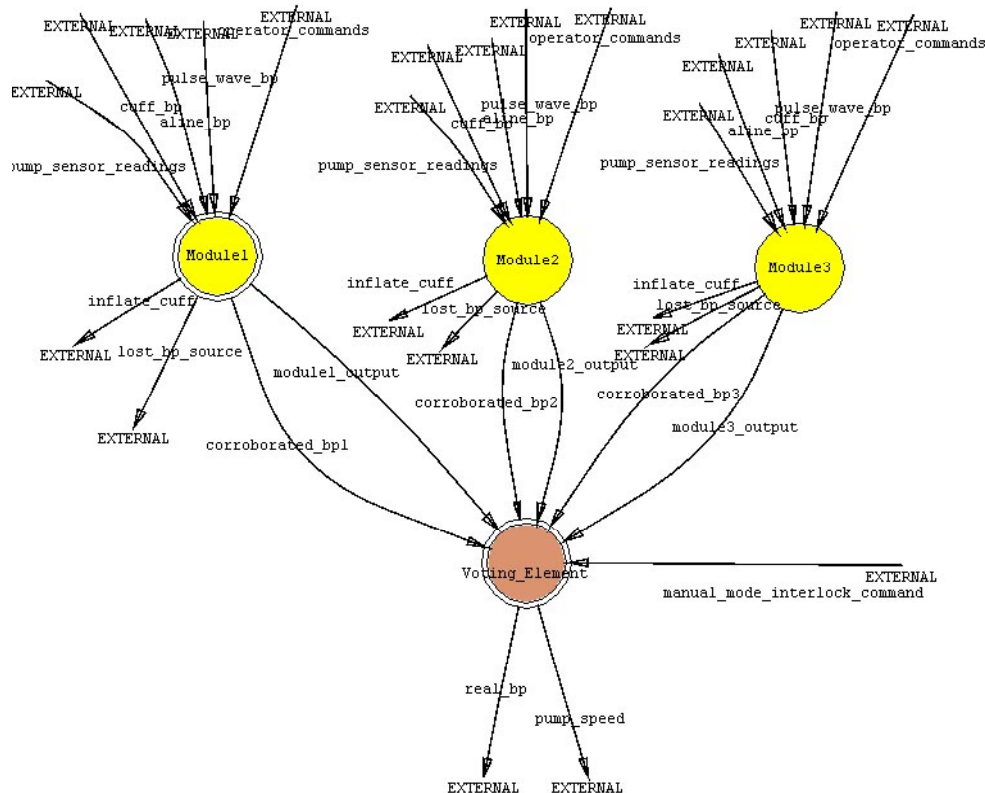


Figure 109 Pump_Control_Module v2

6. Parent Vertex: IO_Module

This module handles the input and output to the CARA display for the benefit of the CARA operator. Note that unlike version 1 (Figure 101), in this version the CARA *display* has been modeled as a decomposable terminator within this module (see Figure 110). The alarm functions have been separated from other display functions to help isolate the safety critical functions. Also note the duplicate *alarm_controller1*. This additional *alarm_controller* would be implemented on the second processor with the *processor_watchdog* so that in case of processor failure, alarms would be raised to the display.

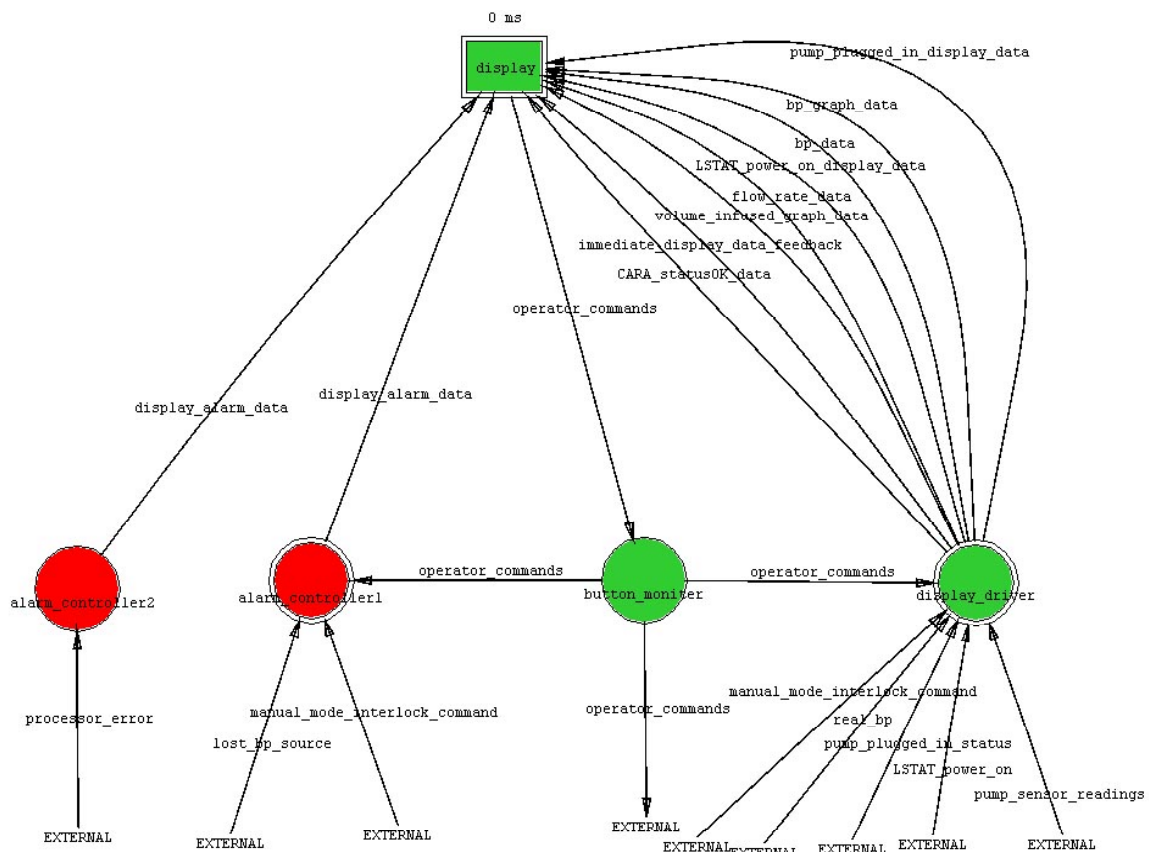


Figure 110 IO Module v2

7. Parent Vertex: Line_Monitor

This module (inside the *Management_Module*) handles and monitors the sensor readings coming from the pump (see Figure 111). In case of any problem readings, a *line_fault* is generated which is immediately sent to the *manual_mode_interlock* (see Figure 108). Determining the proper value of *impedance*, *air_disrupted*, and *back_EMF* readings requires further decomposition of these operators.

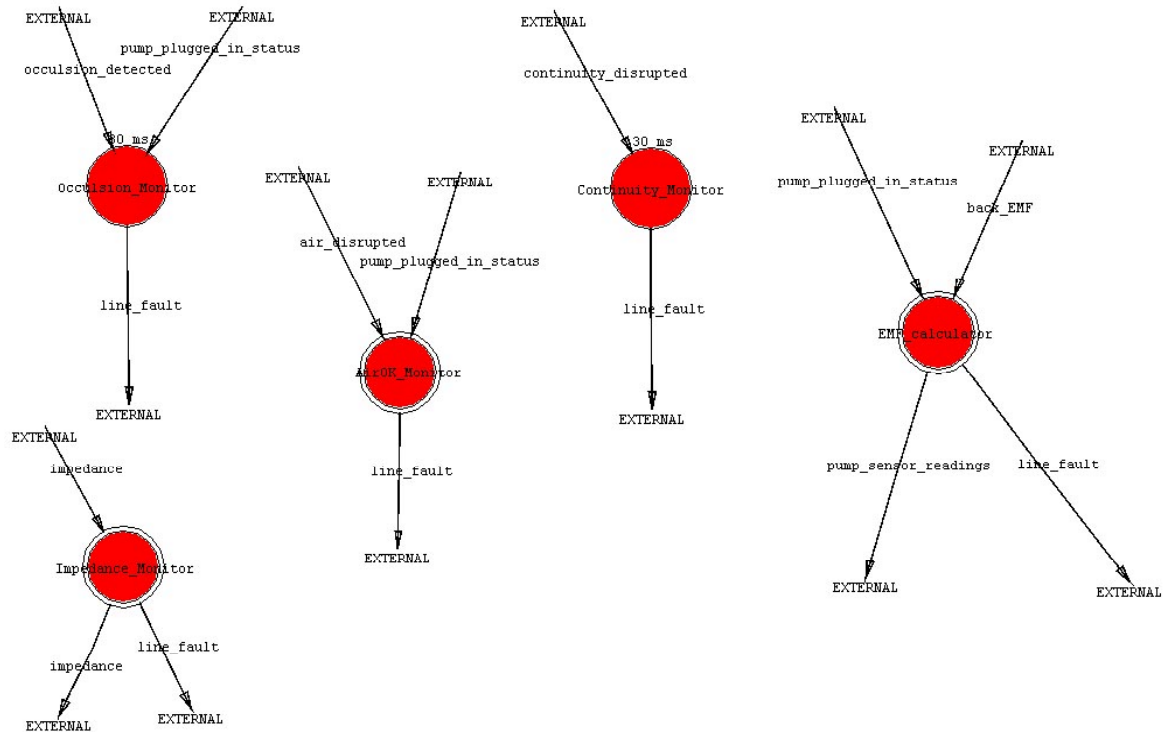


Figure 111 Line_Monitor v2

8. Parent Vertex: Resuscitation_File

This module consists of a series of operators that accept data of a particular type and convert it into *data_for_file* format. The *resuscitation_file* itself is modeled as a looping data stream created by the *Resuscitation_file_Generator* where new additional entries are appended onto the end of the file (see Figure 112 below).

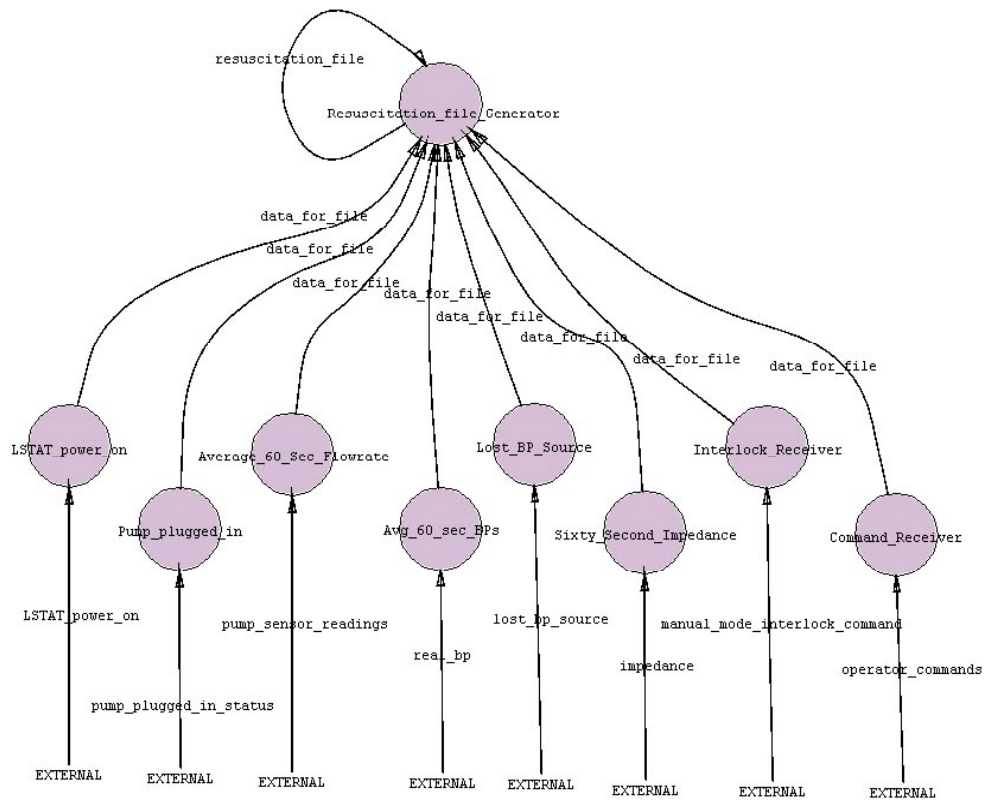


Figure 112 Resuscitation_File v2

9. Parent Vertex: Module1

This parent operator is decomposed into two main functions: first, a blood pressure calculator (*BP_calculator*) responsible for determining which blood pressure to use during further calculations, and second, a *Pump_Speed_Calculator* which determines the appropriate pump command to issue (see Figure 113). Recall that there are two other concurrent modules (*Module2* & 3) (Figure 109) that are performing similar tasks but are utilizing different computing algorithms.

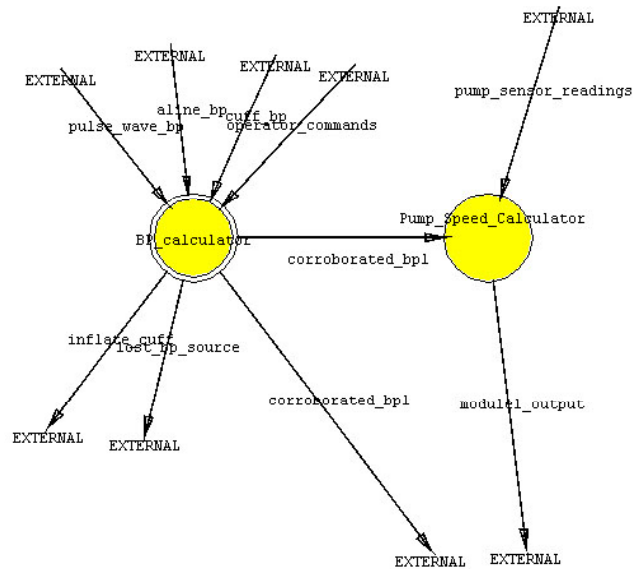


Figure 113 Module1 v2

10. Parent Vertex: Voting_Element

This operator is decomposed into two main sub-operators (see Figure 114). First, the *Vote* operator compares the inputs from *Module1*, 2, 3. If all the data are within a set tolerance of each other, the operator averages the values and outputs the *real_bp* (for the *display* and the *resuscitation_file*) and the *pump_speed*. If two of the inputs are within tolerance and one is outside tolerance, it disregards the value outside tolerance, averages the other two and outputs the average. If all three values are outside tolerance, the *Vote* operator disregards all three values and waits for satisfactory data. The *Terminate_Autocontrol* operator is responsible for returning the system to manual control if the *manual_mode_interlock_command* is invoked.

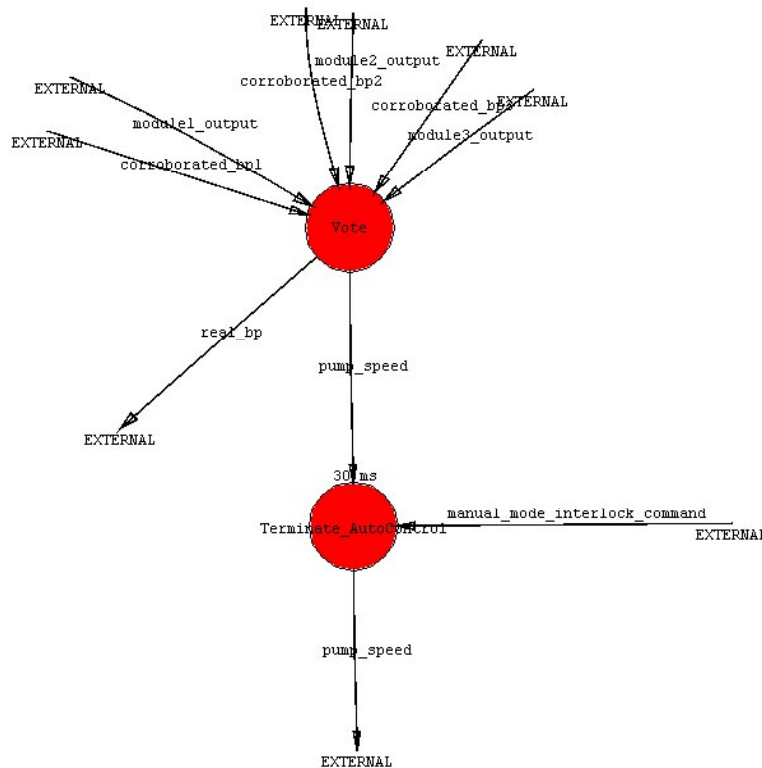


Figure 114 Voting_Element v2

11. Parent Vertex: Alarm_Controller1

This operator accepts alarm generating data streams and outputs alarm data (both text and audio alarm) to the CARA display(see Figure 115 below). A second controller (*alarm_controller2* recall Figure 104) is implemented on the second processor to enable alarm data to be transmitted to the display in the case of processor failure.

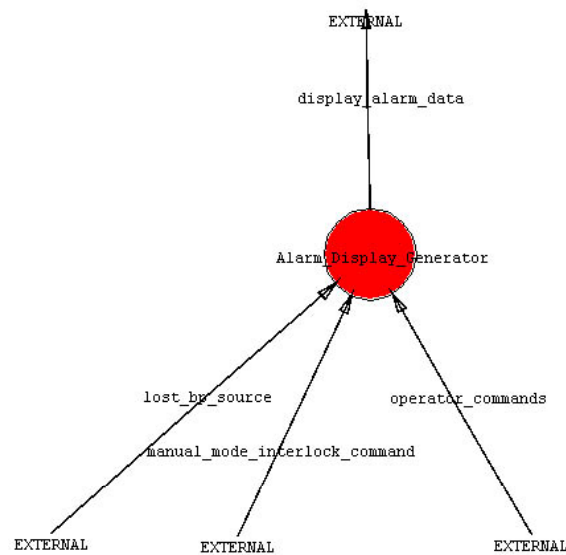


Figure 115 Alarm_Controller1 v2

12. Parent Vertex: Display_Driver

This module contains a series of operators that act as drivers for the information displayed on the CARA *display* (see Figure 116). Some of these operators (the graph operators in particular) maintain aggregate data to send to the display. Also several of the operators function by comparing any new data to old data and only update the display in the case of changes.

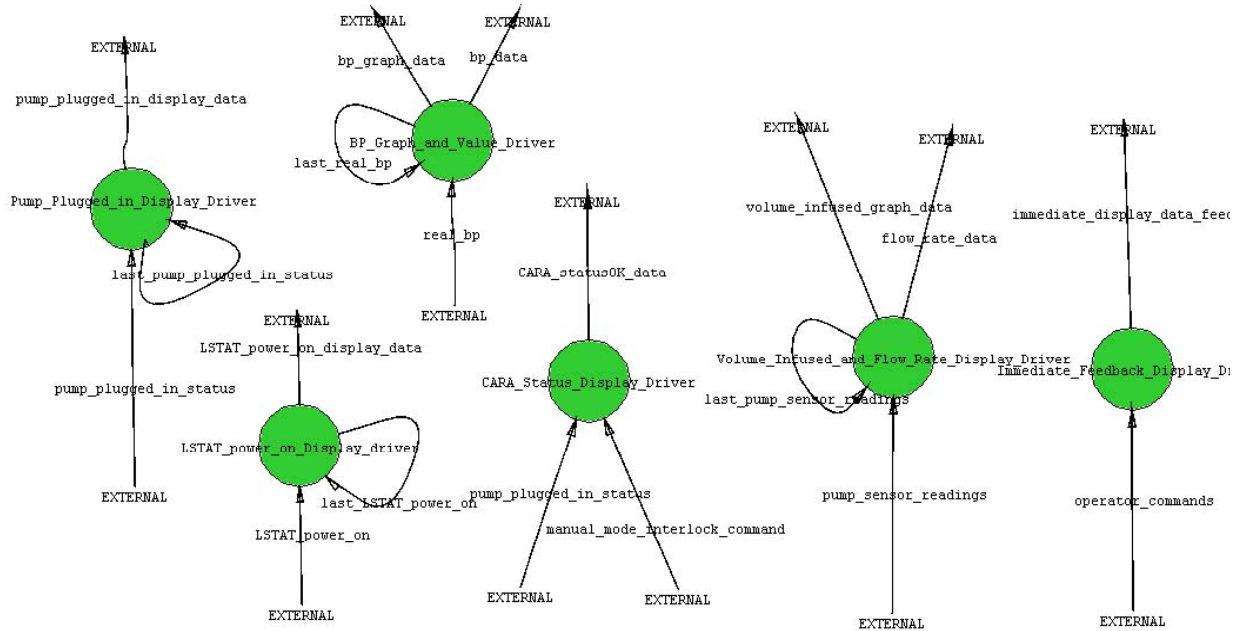


Figure 116 Display_Driver v2

13. Parent Vertex: Display

This module is a decomposed terminator that simulates the functions of the *CARA Display*. Each terminator represents separate sets of data that can be displayed on the *Display* (see Figure 117). This module also simulates user input in the way of *operator_commands* (button pushes) from the *Display*.

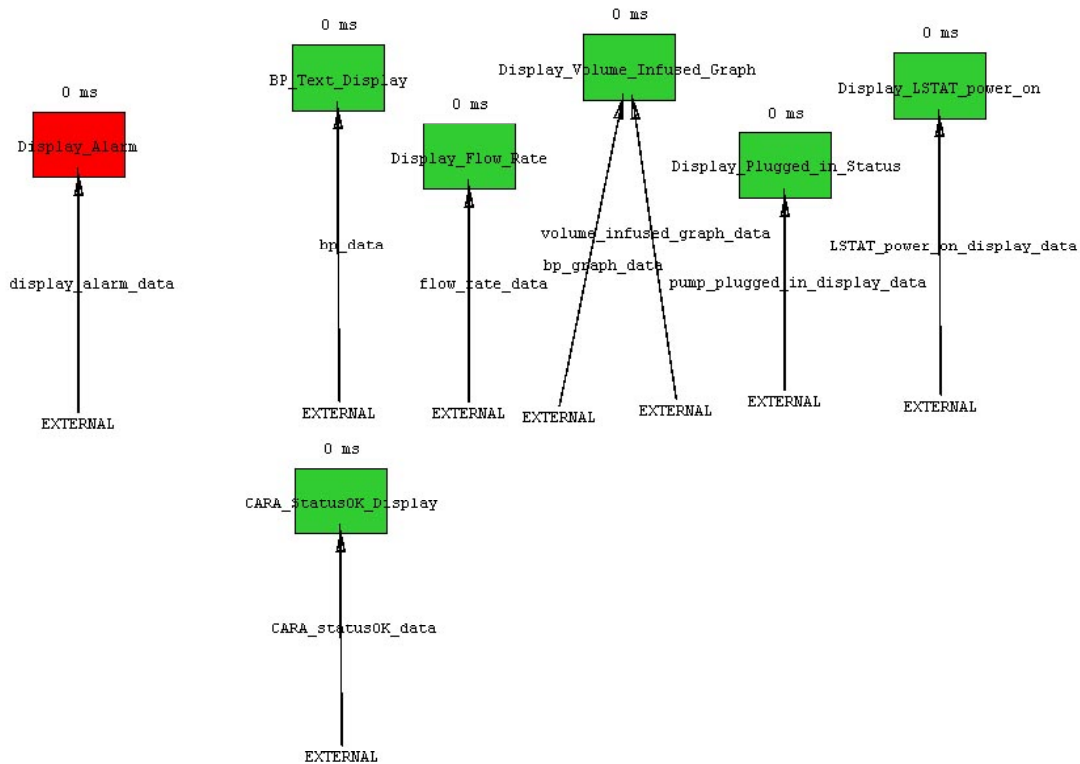


Figure 117 Display v2

14. Parent Vertex: AirOK_Monitor

This operator is decomposed in order to allow a timer to function. If the *Start_Air_Timer* receives input that there is air disruption, then it waits the allotted time to see if air becomes OK. If it does not, then the *Generate_Air_fault* fires indicating a *line_fault*. If the *Start_Air_Timer* does receive good data, then the timer is reset and the fault generator does not fire (see Figure 118 below).

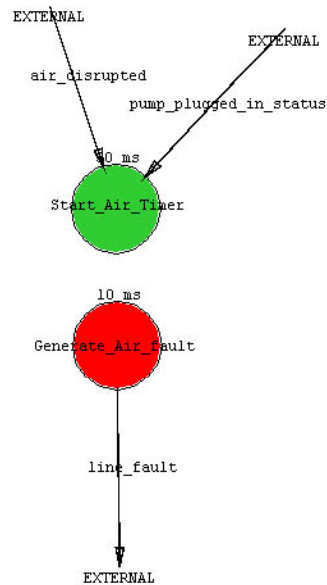


Figure 118 AirOK_Monitor v2

15. Parent Vertex: EMF_calculator

Two major functions are modeled in this module (see Figure 119). First, EMF polling (*Start_EMF_Polling*) takes place so that appropriate *back_EMF* values can be accumulated to calculate both the infuse rate and the total volume infused (*Calculate_Infused_Volume*). These values are later sent to the *display* and the *resuscitation_file* (recall Figure 108 and Figure 111). Secondly, if the *back_EMF* values are unacceptable (absent or out of tolerance), the module waits the appropriate time for acceptable data and then generates a *line_fault* that eventually sets the system back into manual mode.

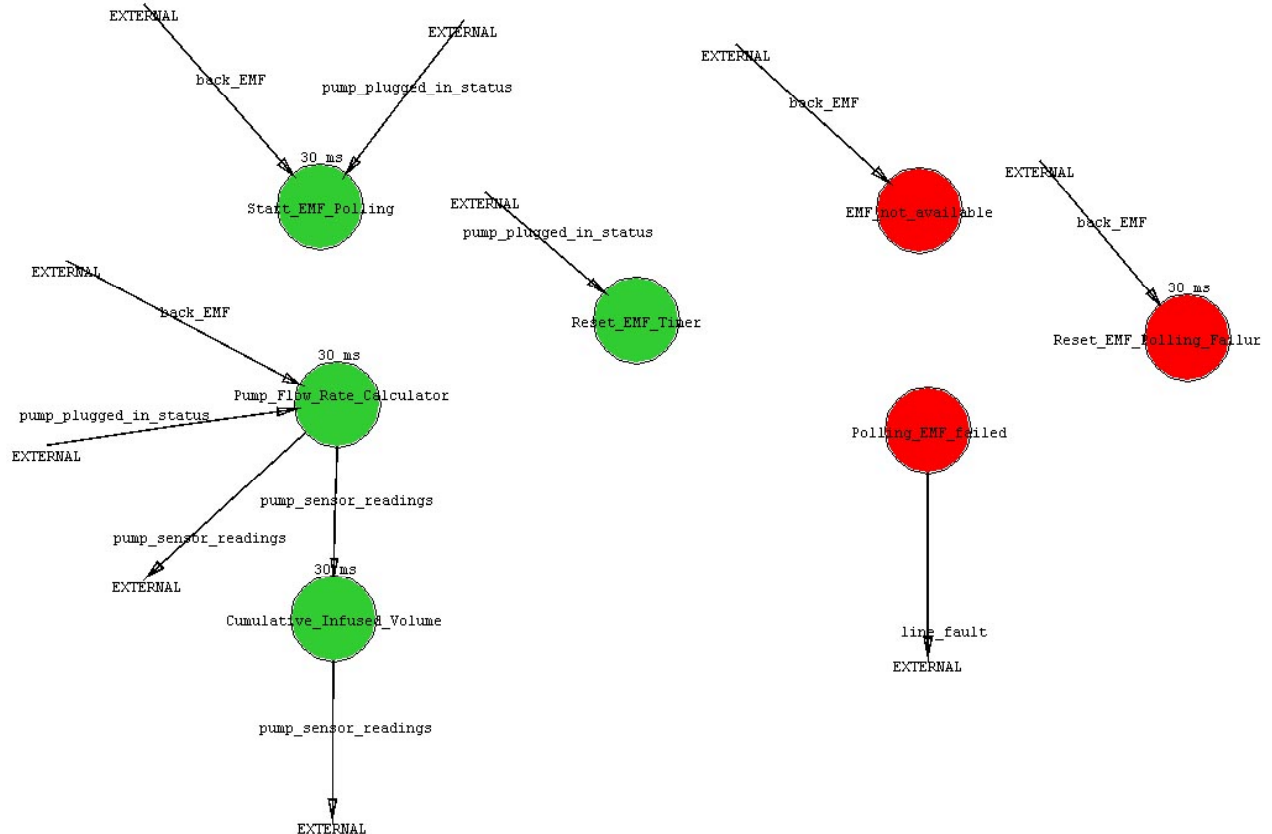


Figure 119 EMF_Calculator v2

16. Parent Vertex: Impedance_Monitor

Much like the *EMF_Calculator*, two major functions are modeled in this module (see Figure 120). First, *impedance* polling takes place (*Start_Impedance_Polling*) so that appropriate *impedance* values can be accumulated to send to the *resuscitation_file*. Secondly, if the *impedance* values are unacceptable (absent or out of tolerance -- *Low_or_No_Impedance*), the module waits the appropriate time for acceptable data and then generates a *line_fault* that eventually sets the system back into manual mode.

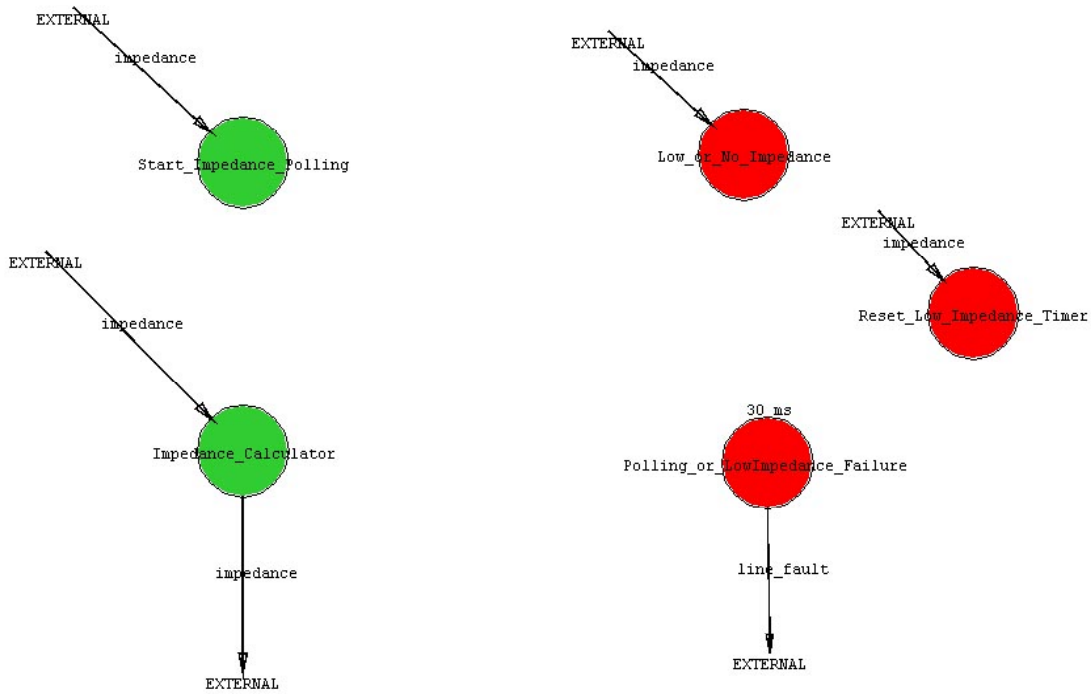


Figure 120 Impedance_Monitor v2

17. Parent Vertex: BP_Calculator

This operator has been decomposed into three operators: first, the *Aline_Corroborator* which attempts to generate a corroborated arterial line blood pressure for future calculation; second, the *Pulse_Wave_Corroborator* which attempts to generate a corroborated pulse wave blood pressure for future calculation; and finally, a *BP_Priority_Calculator* which given the blood pressures available (arterial line, pulse wave, and cuff) generates the blood pressure that will be used (*corroborated_bp1*) based on a blood pressure priority scheme (see Figure 121).

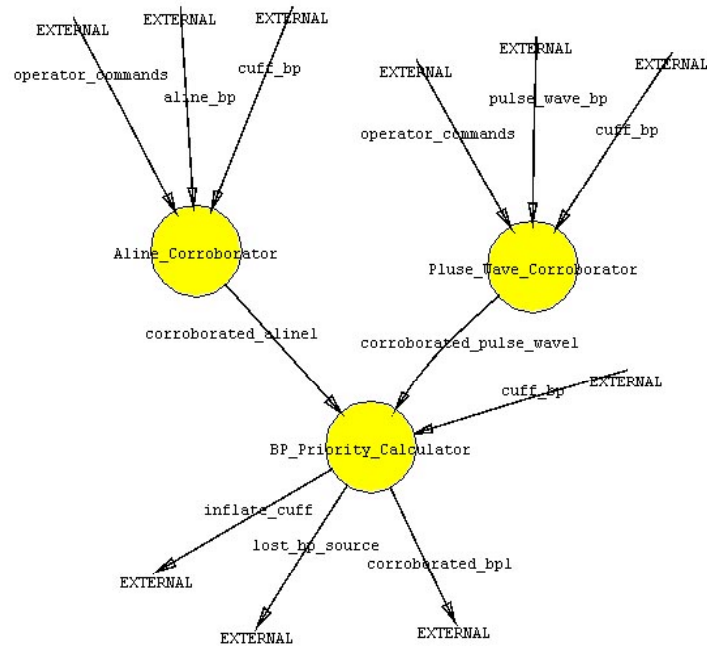


Figure 121 BP_Calculator v2

LIST OF REFERENCES

- [AKAO90] Akao, Y., *Quality Function Deployment: Integrating Customer Requirements into Product Design*, Tamagawa University, Japan translated by Glenn H. Mazur and Japan Business Consultants, Ltd., Productivity Press, Cambridge, Mass, 1990.
- [ANDE94] Anderson, K., Taylor, R., and Whitehead, J., "Chimera: Hypertext for Heterogeneous Software Environments," *Proceedings of the European Conference n Hypermedia Technology (ECHT '94)*, Edinburgh, Scotland, September 1994, pp. 94-107.
- [ARCA95] Arcadia Consortium, *The Collected Arcadia Papers, Volume 1: Software Engineering Environment Infrastructure*, 2nd Ed., 1995.
- [BADR93] Badr, S., "A Model and Algorithms for a Software Evolution Control System," *Ph.D. Dissertation*, Computer Science Department, Naval Postgraduate School, Monterey, CA, December, 1993.
- [BERG89] Berge, C., *Hypergraphs: Combinatorics of Finite Sets*, North-Holland, 1989.
- [BERZ91] Berzins, V. and Luqi, *Software Engineering with Abstractions*, Addison-Wesley, 1991.
- [BETT90] Betts, M., "QFD Integrated with Software Engineering," *Transactions of the Second Symposium on Quality Function Deployment*, Novi MI, 18-19 June 1990, pp. 442-459.
- [BOEH95] Boehm, B., Madachy, R., and Selby, R., "Cost Models for Future Software Life Cycle Processes: COCOMO 2.0," *Annals of Software Engineering Special Volume on Software process and Product Measurement*, J.C. Baltzer AG, Science Publishers, Amsterdam, 1995.
- [BORI86] Borison, E., "A Model of Software Manufacture," *Advanced Programming Environments: Proceedings of an International Workshop*, Eds. R. Conradi, T. M. Didriksen, and D. H. Wanvik, Trondheim Norway, June 1986, pp. 197-220.
- [BOUD88] Boudier, G., Gallo, F., Minot, R., and Thomas, I., "An Overview of PCTE and PCTE+," *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SESPSDE)*, 1988.

- [BROW92] Brown, A.W. and McDermid, J.A., "Learning from IPSE's Mistakes," *IEEE Software*, Vol 9, Issue: 2, March 1992, pp. 23 -28.
- [BROW93] Brown, A.W., "An Examination of the current state of IPSE Technology," *Proceedings of the 15th International Conference on Software Engineering*, 17-21 May 1993, pp. 338-347.
- [CAMP63] Campbell, D. T. and Stanley, J. C., *Experimental and Quasi-Experimental Designs for Research*, Houghton Mifflin Company, Boston, 1963.
- [COHE95] Cohen, L., *Quality Function Deployment: How to Make QFD Work for You*, Addison-Wesley, 1995.
- [CLAU88] Clausing, D., "Quality Function Deployment," in Ryan, N. E., ed., *Taguchi Methods and QFD*, American Supplier Institute Inc., Dearborn, MI, 1988.
- [CLOM03] Clomera, A., " Extending the Computer-Aided Software Evolution System (CASES) with Quality Function Deployment (QFD)" *Masters Thesis*, Computer Science Department, Naval Postgraduate School, Monterey, CA, June 2003
- [CORM91] Cormen, T., Leiserson, C., and Rivest, R., *Introduction to Algorithms*, 4th Printing, MIT Press, 1991.
- [CRAN99] Cranefield, S. and Purvis, M., "UML as an Ontology Modelling Language", *Proceedings of the IJCAI'99 Workshop on Intelligent Information Integration*, Sweden, 1999.
- [CRAN01] Cranefield, S., Haustein, S., and Purvis, M.,. "UML-Based Ontology Modelling for Software Agents," *Proceedings of Ontologies in Agent Systems Workshop*, Agents 2001, Montreal, Canada, 2001.
- [CZAR00] Czarnecki, K. and Eisenecker, U., *Generative Programming Methods, Tools, and Applications*, Addison-Wesley, 2000.
- [DAML03] "DARPA Agent Markup Language", <http://www.daml.org/>, 3 March 2003.
- [DEAN92] Dean, E. B., "Quality Function Deployment for Large Systems", *Proceedings of the 1992 International Engineering Management Conference*, Eatontown, NJ, USA, 25-28 October 1992.

- [DEVO00] Devore, J. L., *Probability and Statistics for Engineering and the Sciences*, 5th Edition, Duxbury--Brooks/Cole, 2000.
- [FITZ03] Fitzgerald, B., Russo, N., and O’Kane, T., “Software Development Method Tailoring at Motorola,” *Communications of the ACM*, Vol. 46, No. 4, April 2003, pp. 65-70.
- [GANG00] Ganguly, P. and Ray, P., “Software Interoperability of Telemedicine Systems: A CSCW Perspective,” *Proceedings of 7th International Conference on Parallel and Distributed Systems*, 2000, pp. 349-356.
- [GAOT98] General Accounting Office Testimony: Dillingham, G. L., “Air Traffic Control: Evolution and Status of FAA’s Automation Program,” *GAO/T-RCED/AIMD-98-85*, 5 March 1998.
- [GEYE00] Geyer, L., “Feature Modeling Using Design Spaces,” *Proceedings of 1st German Workshop on Product Line Software Engineering*, Kaiserslautern, Germany, November 2000.
- [GRUB95] Gruber, T. R., “Toward Principles for the Design of Ontologies Used for Knowledge Sharing,” *International Journal of Human-Computer Studies*, Vol. 43, 1995, pp. 907-928.
- [GRUN95] Grüninger, M., and Fox, M.S., "Methodology for the Design and Evaluation of Ontologies", *Proceedings of the Workshop on Basic Ontological Issues in Knowledge Sharing*, IJCAI-95, Montreal, 13 April 1995.
- [HAAG96] Haag, S., Raja, M. K., and Schkade, L. L., "Quality Function Deployment Usage in Software Development," *Communications of the ACM*, Vol. 39, No. 1, 1996, pp. 41-49.
- [HARN99a] Harn, M., Berzins, V., Luqi, and Kemple, W., "Evolution of C4I Systems," *Proceedings of 1999 Command and Control Research and Technology Symposium*, United States Naval War College, Newport, Rhode Island, June 29 - July 1, 1999, pp.1361-1380.
- [HARN99b] Harn, M., Berzins, V., and Luqi, "Software Evolution Process via a Relational Hypergraph Model," *Proceedings of IEEE/IEEJ/JSAI International Conference on Intelligent Transportation Systems*, Tokyo, Japan, October 5-8, 1999, pp. 599-604.
- [HARN99c] Harn, M., "Computer-Aided Software Evolution Based on Inferred Dependencies" *Ph.D. Dissertation*, Computer Science Department, Naval Postgraduate School, Monterey, CA, 1999.

- [HASN03] Hasni, N., "Computer-Aided Software Evolution Based on Inferred Dependencies" *Masters Thesis*, Computer Science Department, Naval Postgraduate School, Monterey, CA, March 2003.
- [HAUS88] Hauser, J. R. and Clausing, D., "The House of Quality," *The Harvard Business Review*, May-June 1988, No. 3, pp 63-73.
- [HEIM92] Heimbigner, D., "Experiences With an Object Manager for a Process-Centered Environment," *Proceedings of the 18th Very Large Data Bases Conference (VLDB92)*, Vancouver, British Columbia, Canada, 1992.
- [HRON93] Hrones, J. A. Jr., Jedrey, B. C. Jr., and Zaaf, D., "Defining Global Requirements with Distributed QFD," *Digital Technical Journal*, Vol. 5, No. 4, Fall 1993, pp. 36-46.
- [IBRA96] Ibrahim, O. M., "A Model and Decision Support Mechanism for Software Requirements Engineering," *Ph.D. Dissertation*, Computer Science Department, Naval Postgraduate School, Monterey, CA, 1996.
- [KADI92a] Kadia, R., "Lessons from the Arcadia Project", *Proceedings of the Software Technology Conference*, Los Angeles, CA, April 28-30, 1992.
- [KADI92b] Kadia, R., "Issues Encountered in Building a Flexible Software Development Environment: Lessons from the Arcadia Project", *Proceedings of the 5th ACM SIGSOFT Symposium on Software Development Environments*, Tyson's Corner, VA, 1992, pp. 169-180.
- [KOGU02] Kogut, P., Cranefield, S., Hart, L., Dutra, M., Baclawski, K., Kokar, M., and Smith, J., "UML for Ontology development," *Knowledge Engineering Review*, Vol. 17, Issue 1, March 2002, pp. 61-64.
- [KRUC96] Kruchten, P., "A Rational Development Process," *CrossTalk*, 9(7), July 1996, STSC, Hill AFB, UT, pp. 11-16.
- [KUNZ70] Kunz, W., Rittel, H. W. J., "Issues as Elements of Information Systems," *Working Paper 0131*, Institut für Grundlagen der Planung: Universität Stuttgart, 1970.
- [LAMI95] Lamia, W. M., "Integrating QFD with Object Oriented Software Design Methodologies," *Transactions from the Seventh Symposium on Quality Function Deployment*, Novi MI, 11-13 June 1995, pp. 417-434.
- [LAWL03] Lawler, G., *Guidelines for Updating Protégé XML Schemas*, Internal Naval Postgraduate White Paper, February 2003.

- [LEFF00] Leffingwell, D. and Widrig, D., *Managing Software Requirements: A Unified Approach*, Addison-Wesley, 2000.
- [LEHC99] Le, H. C. T., "Design of a Persistence Server for the Relational Hypergraph Model," *Masters Thesis*, Naval Postgraduate School, December 1999.
- [LEHM69] Lehman, M., "The Programming Process," *IBM Research Report RC 2722*, IBM Research Center, Yorktown Heights, NY, September 1969.
- [LEHM87] Lehman, M.M. and Turski, W.M., "Essential Properties of IPSEs," *Software Engineering Notes*, Vol. 12, Issue 1, 1987, pp. 52-55.
- [LEHM91] Lehman, M., "Software Engineering, the Software Process and their Support," *Software Engineering Journal*, Vol. 6, Issue 5, September 1991, pp. 243-258.
- [LEHM97] Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E., and Turski, W.M., "Metrics and Laws of Software Evolution -- The Nineties View," *Proceedings of the 4th International Software Metrics Symposium*, 5-7 November 1997, pp. 20-32.
- [LEHM98] Lehman, M., "Software's Future: Managing Evolution," *IEEE Software*, Vol. 15, Issue 1, January/February 1998, pp. 40-44.
- [LEHM00] Lehman, M.M., and Ramil, J.F., "Towards a Theory of Software Evolution -- And its Practical Impact," *Proceedings of the International Symposium on Principles of Software Evolution*, 2000, pp. 2-11.
- [LENC01] Lenci, A., "Building an Ontology for the Lexicon: Semantic Types and Word Meaning," in Jensen and Skadhauge (eds.), pp. 103-120, 2001, http://www.ontoquery.dk/publications/docs/Building_an_Ontology.doc, 03/02/03.
- [LIEN78] Lientz, B. P., Swanson, E. B., and Tompkins, G. E., "Characteristics of Application Software Maintenance," *Communications of the ACM*, Vol. 21, No. 6, June 1978.
- [LIUX00] Liu, X. F., "Software Quality Function Deployment," *IEEE Potentials*, Vol. 19, Issue 5, December 2000/January 2001, pp. 14-16.
- [LUQI88] Luqi and Ketabchi, M., "A Computer-Aided Prototyping System", *IEEE Software*, 5(2), pp. 66-72, 1988.

- [LUQI89] Luqi, "Software Evolution Through Rapid Prototyping," *IEEE Computer*, Vol. 22, Issue 5, May 1989, pp. 13-25.
- [LUQI90] Luqi, "A Graph Model for Software Evolution," *IEEE Trans. on Software Engineering*, Vol. 16, No. 8, August 1990, pp. 917-927.
- [LUQI91a] Luqi, "Computer-Aided Software Prototyping", *IEEE Computer*, pp. 111-112, September 1991.
- [LUQI91b] Luqi, Steigerwald, R., Hughes, G., and Berzins, V., "CAPS as a Requirement Engineering Tool," *Proceeding of Requirements Engineering and Analysis Workshop*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, March 12-14, 1991, pp. 1-8.
- [LUQI96] Luqi, "System Engineering and Computer-Aided Prototyping", *Journal of Systems Integration - Special Issue on Computer Aided Prototyping*, vol. 6, No. 1, pp. 15-17, 1996.
- [LUQI02] Luqi, Shing, M., Berzins, V., Puett, J., et. al. "Computer Aided Prototyping for the Infusion Pump Computer Assisted Resuscitation Algorithm (CARA) Software," Naval Postgraduate School Technical Report NPS-SW-02-004, 2002.
- [LUQI03] Luqi, Shing, M., Puett, J., Berzins, v., et. al. "Comparative Rapid Prototyping, A Case Study," *Proceedings of the 14th IEEE International Workshop on Rapid Systems Prototyping*, San Diego, California, June 2003.
- [MCBR02] McBreen, P., *Software Craftsmanship: The New Imperative*, Addison-Wesley, 2002.
- [MEYE97] Meyer, B., *Object-Oriented Software Construction*, 2nd Ed., Prentice Hall PTR, 1997.
- [MUSE98] Musen, M.A., "Domain Ontologies in Software Engineering: Use of Protégé with the EON Architecture," *Methods of Information in Medicine*, Vol. 37, No. 4-5, 1998, pp. 540-550.
- [NOYN01] Noy, N.F. and McGuinness, D. L., "Ontology Development 101: A Guide to Creating Your First Ontology," *Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880*, March 2001.
- [PARI83] Parikh, G. and Zvegintzov, N., *Tutorial on Software Maintenance*, IEEE Computer Society Press, 1983.

- [PITO97] Pitoura, E., "Providing Database Interoperability through Object-Oriented Language Constructs", *Journal of Systems Integration*, Volume 7, No. 2, August 1997, pp. 99-126.
- [PRES01] Pressman, R., *Software Engineering: A Practitioner's Approach*, 5th Ed., McGraw Hill, Boston, 2001.
- [PROT03a] Protégé Project Homepage, <http://protege.stanford.edu/index.html>, 2 March 2003.
- [PROT03b] Protégé-2000 User's Guide, <http://protege.stanford.edu/publications/UserGuide.pdf>, 2 March 2003.
- [PUET02a] Puett, J., "Holistic Framework for Establishing Interoperability of Heterogeneous Software Development Tools and Models," *Proceedings of the 24th International Conference on Software Engineering (ICSE02)*, Orlando Florida, May 2002, pp 729-730.
- [PUET02b] Puett, J., "Holistic Framework for Establishing Interoperability of Heterogeneous Software Development Tools and Models," *Proceedings of the ICSE-02 Doctoral Symposium*, Orlando Florida, 21 May 2002.
- [RATI98] Rational Software Corporation, *Rational Unified Process: Best Practices for Software Development Teams*, TP-026A, rev. November 98.
- [RATI01] Rational Software Corporation, *Rational RequisitePro User's Guide*, Version 2002.05.00, 2001.
- [RATI03] Rational Software Corporation, "Rational Unified Process Homepage," <http://www.rational.com/products/rup/prodinfo.jsp>, 3 March 2003.
- [ROSE02] *Rational Rose® User's Help*, Release Version 2002.05.00, 2002.
- [SAAT80] Saaty, T. L., *The Analytic Hierarchy Process*, McGraw-Hill, 1980.
- [SHAR91] Sharkey, A. I., "Generalized Approach to Adapting QFD for Software," *Transactions of the Third Symposium on Quality Function Deployment*, Novi MI, 24-25 June 1991, pp. 380-416.
- [SOMM01] Sommerville, I., *Software Engineering*, 6th Edition, Addison-Wesley, 2001.
- [SOWA00] Sowa, J. F., *Knowledge Representation: Logical, Philosophical, and Computational Foundations*, Pacific Grove Brooks/Cole, 2000.

- [STAN00] *STANAG 5048: The Minimum Scale of Connectivity for Communications and Information Systems for NATO Land Forces*, Edition 5, Promulgated 16 February 2000 by NC3B Sub-Committee AC/322 SC/1.
- [STEV46] Stevens, S. S., "On the Theory of Scales of Measurement," *Science*, New Series, Vol. 103, No. 2684, 7 June 1946, pp. 677-680.
- [STEV51] Stevens, S. S., "Mathematics, Measurement, and Psychophysics," in *Handbook of Experimental Psychology*, ed. S. S. Stevens, John Wiley, New York, 1951.
- [SUTT95] Sutton, S., Hiembigner, D., and Osterweil, L., "APPL/A: A Language for Software-Process Programming," in *The Collected Arcadia Papers, Volume 1: Software Engineering Environment Infrastructure*, 2nd Ed., 1995, pp. 91-132.
- [SWEB01] *Software Engineering Body of Knowledge*, Stone man Version 0.9, Executive Editors Alain Abran and James Moore, February 2001.
- [TARR93] Tarr, P., and Clarke, L., "PLEIADES: An Object management System for Software Engineering Environments," *Proceeding on foundations of Software Engineering*, Los Angeles, California, 1993, pp. 56-70.
- [TAYL94] Taylor, R. Nies, K, Bolcer, G. MacFarlane, C. Johnson, G. and Anderson, K., "Supporting Separations of Concerns and Concurrency in the Chiron-1 User Interface System," 11 March 1994, in *The Collected Arcadia Papers, Volume 1: Software Engineering Environment Infrastructure*, 2nd Ed., 1995, pp. 19-68.
- [THAC90] Thackery, R. and Van Treeck, G., "Applying Quality Function Deployment for Software Product Development," *Journal of Engineering Design*, Vol. 1, No. 4, 1990, pp. 389-410.
- [TOGE03] TogetherSoft Homepage, <http://www.togethersoft.com/>, 5 March 2003.
- [USCH96] Uschold, M. and Gruninger, M., "Ontologies: Principles, Methods and Applications," *Knowledge Engineering Review*, Vol. 11, No. 2, June 1996.
- [USCH98] Uschold, M., King, M., Moralee, S., and Zorgios, Y., "The Enterprise Ontology," *Knowledge Engineering Review*, Vol 13, Issue 1, Cambridge University Press, March 1998, pp. 31-89.
- [VELL93] Velleman, P. F. and Wilkinson, L., "Nominal, Ordinal, Interval, and Ratio Typologies are Misleading," *The American Statistician*, Vol. 47, No. 1, February 1993, pp. 65-72.

- [WRAI01a] WRAIR Dept. of Resuscitative Medicine, *Narrative Description of the CARA software*, Proprietary Document, WRAIR, Silver Spring, MD, January 2001.
- [WRAI01b] WRAIR Dept. of Resuscitative Medicine, *CARA Pump Control Software Questions, Version 6.1*, Proprietary Document, WRAIR, Silver Spring, MD, January 2001.
- [WRAI01c] WRAIR Dept. of Resuscitative Medicine, *CARA Tagged Requirements, Increment 3, Version 1.2*, Proprietary Document, WRAIR, Silver Spring, MD, March 2001.
- [YOUN01] Young, P., Ge Jun, Berzins, V., Luqi, "Using an Object Oriented Model for Resolving Representational Differences between Heterogeneous Systems," *Proceedings of the Monterey Workshop 2001*, June 2001.
- [YOUN02a] Young, P., Berzins, V., Ge Jun, Luqi, "Using an Object Oriented Model for Resolving Representational Differences between Heterogeneous Systems," *Proceedings of the ACM Symposium on Applied Computing SAC2002*, Madrid Spain, March 2002.
- [YOUN02b] Young, P., "Heterogeneous Software System Interoperability Through Computer-Aided Resolution Of Modeling Differences," *PhD Dissertation*, Naval Postgraduate School, June 2002.
- [ZULT90] Zultner, R. E., "Software Quality Deployment: Adapting QFD to Software," *Transactions of the Second Symposium on Quality Function Deployment*, Novi MI, 18-19 June 1990, pp. 132-149.
- [ZULT92] Zultner, R. E., "Quality Function Deployment (QFD) for Software: Structured Requirements Exploration," in Schulmeyer, G. G. and J. I. McManus, ed., *Total Quality Management for Software*, Van Nostrand Reinhold, New York NY, 1992.
- [ZULT93] Zultner, R. E., "TQM for Technical Teams," *Communications of the ACM*, Vol. 36, No. 10, 1993, pp. 79-91.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California
3. Professor Luqi
Naval Postgraduate School
Monterey, California
4. Professor James B. Michael
Naval Postgraduate School
Monterey, California
5. Professor Craig W. Rasmussen
Naval Postgraduate School
Monterey, California
6. Professor Man-Tak Shing
Naval Postgraduate School
Monterey, California
7. Dr. Nelson Ludlow
Mobilisa, Incorporated
Port Townsend, Washington
8. Mr. Douglas Lange
Space and Naval Warfare Systems Center
San Diego, California
9. Dr. David Hislop
U.S. Army Research Office
Research Triangle Park, NC
10. Mr. Paul L. Jones
U.S. Food and Drug Administration
Rockville, Maryland

11. Dr. Stephen A. Van Albert
Walter Reed Army Institute of Research
Washington D. C.
12. Dr. Frederick J. Pearce
Walter Reed Army Institute of Research
Washington D. C.
13. Professor Valdis Berzins
Naval Postgraduate School
Monterey, California
14. Professor Peter Denning
Naval Postgraduate School
Monterey, California
15. Professor David Floodeen
Naval Postgraduate School
Monterey, California
16. LCDR Christopher Williamson
HELLANTISUBRON EIGHT
Unit 25174, FPO/AP 96601-5709
17. CAPT Paul E. Young
U.S. Naval Academy
Annapolis, Maryland
18. LTC Joseph F. Puett III
Naval Postgraduate School
Monterey, California